National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Performance Investigation of various Microservice Architectures

Diploma Thesis

of

alexandros kyriakakis

Supervisor: Vassilios Vescoukis
Professor

Athens, July 2023

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Performance Investigation of various Microservice Architectures

Diploma Thesis

of

ALEXANDROS KYRIAKAKIS

Supervisor: Vassilios Vescoukis
Professor

Approved by the examination committee on July 10th.

(Signature)	*(Signature)*	*(Signature)*
.....	.....	.....
Vassilios Vescoukis	Tsanakas Panayiotis	Aris Pagourtzis
Professor	Professor	Professor

Athens, July 2023

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

(Signature)

. . . . . .
Alexandros Kyriakakis

10 July 2023

Περίληψη

Η αρχιτεκτονική λογισμικού αποτελεί ένα σημαντικό μέρος των σύγχρονων ψηφιακών συστημάτων, επηρεάζοντας σημαντικά την απόδοση, την κλιμακωσιμότητα και τη συντηρησιμότητα τους. Τα τελευταία χρόνια, επιχειρήσεις και εταιρείες αντιμετωπίζουν αυξανόμενες προκλήσεις στον προσδιορισμό και την εφαρμογή της βέλτιστης αρχιτεκτονικής λογισμικού, για τις μοναδικές απαιτήσεις τους. Αυτή η περίπλοκη διαδικασία λήψης αποφάσεων μπορεί να επηρεάσει σημαντικά την επιτυχία των συστημάτων λογισμικού τους, καθώς και τις συνολικές επιχειρηματικές λειτουργίες.

Στην παρούσα εργασία, μετά από μια συγκριτική ανάλυση κυρίαρχων αρχιτεκτονικών στυλ λογισμικού - συγκεκριμένα των μονολιθικών (monolithic), των αρχιτεκτονικών προσανατολισμού υπηρεσιών, (Service-Oriented Architectures/SOA) και των μικροϋπηρεσιών (microservices) - δίνεται ιδιαίτερη έμφαση στην ολοένα και πιο δημοφιλή αρχιτεκτονική μικροϋπηρεσιών, ρίχνοντας φως στα οφέλη, τις προκλήσεις και τις τεχνικές επικοινωνίας της, ειδικότερα, την ενορχήστρωση και τη χορογραφία.

Για να δοθεί πρακτικό πλαίσιο σε αυτές τις θεωρητικές έννοιες, η παρούσα εργασία αναλαμβάνει μια μελέτη περίπτωσης, χρησιμοποιώντας μια σειρά από τεχνικά εργαλεία, όπως το Docker, το Prometheus και το Grafana, και διαμεσολαβητές μηνυμάτων, όπως το RabbitMQ και το Redpanda. Η μελέτη διερευνά σχολαστικά πέντε διαφορετικές αρχιτεκτονικές, διανύοντας όλες τις φάσεις της ανάπτυξης και των δοκιμών αυτών. Εξετάζει επιπλέον τα ευρήματα, οδηγώντας σε περιεκτικά συμπεράσματα. Η έρευνα αυτή παρέχει πολύτιμες γνώσεις σε όσους έρχονται αντιμέτωποι με αποφάσεις αρχιτεκτονικής λογισμικού και συμβάλλει στην ευρύτερη κατανόηση του ρόλου της αρχιτεκτονικής λογισμικού, στην επιτυχή ανάπτυξη λογισμικού και στην επιχειρηματική στρατηγική.

Λέξεις Κλειδιά

Αρχιτεκτονική Λογισμικού, Μικροϋπηρεσίες, Ενορχήστρωση, Χορογραφία, RabbitMQ, RedPanda, Prometheus, Grafana, Docker

Abstract

Software architecture stands as a critical aspect in the modern landscape of digital systems, strongly influencing their performance, scalability, and maintainability. In recent years, businesses and organizations have faced growing challenges in identifying and implementing the optimal software architecture for their unique requirements. This intricate decision-making process can significantly impact the success of their software projects, as well as overall business operations.

In this thesis, after a comparative analysis of prevalent software architectural styles - namely, monolithic, Service Oriented Architecture (SOA), and microservices - a particular emphasis is placed on the increasingly popular microservices architecture, shedding light on its benefits, challenges and communication techniques, specifically, orchestration and choreography.

To bring practical context to these theoretical concepts, this thesis undertakes a case study, employing a range of technical tools such as Docker, Prometheus and Grafana and message brokers, such as RabbitMQ and Redpanda. The study meticulously explores five distinct architectures, traversing through every phase of development and testing. It further examines the findings, leading to comprehensive conclusions. This investigation provides valuable insights to those grappling with software architecture decisions and contributes to the broader understanding of software architecture's role in successful software development and business strategy.

Keywords

Software Architecture, Microservices, Orchestration, Choreography, RabbitMQ, Redpanda, Prometheus, Grafana, Docker

Acknowledgements

After completing my thesis, my five-year trip to the School of Electrical and Computer Engineering (ECE) at the National Technical University of Athens (NTUA) comes to an end. This would not be possible without my family and friends, who supported me throughout my undergraduate studies.

I would like to express my gratitude to my professor and supervisor Mr Vassilios Vescoukis, for the opportunity to work and get hands-on experience on this subject, but also for his valuable guidance, feedback and excellent cooperation during my thesis.

In addition, I wish to acknowledge the help provided by the ECE NTUA Post Graduate Konstantina Freri, who assisted me throughout my thesis, to understand the subject in depth.

Finally, I wish to extend my special thanks to all the people who were part of this journey at the National Technical University of Athens.

Athens, July 2023

Alexandros Kyriakakis

Εκτεταμένη Περίληψη

Αρχιτεκτονική Λογισμικού

Η αρχιτεκτονική λογισμικού είναι ένα κρίσιμο στοιχείο της τεχνολογίας λογισμικού, ειδικά στη διαχείριση της ανάπτυξης και συντήρησης συστημάτων μεγάλης κλίμακας. Αναφέρεται στο σύνολο των δομών που χρειάζονται για να αναλύσουμε ένα σύστημα, το οποίο περιλαμβάνει τα στοιχεία του λογισμικού, τις σχέσεις τους και τις ιδιότητές τους. Η αρχιτεκτονική λογισμικού μπορέι να χαρακτηριστεί και ως γέφυρα μεταξύ των απαιτήσεων του συστήματος και της υλοποίησης. Η αρχιτεκτονική αποφασίζεται κατά τα αρχικά στάδια της ανάπτυξης και επηρεάζει την υλοποίηση λειτουργικών αναγκών, μη λειτουργικών απαιτήσεων και επιχειρηματικών στόχων. Έτσι, η σωστή επιλογή της αρχιτεκτονικής είναι κρίσιμη για την επιτυχία ενός έργου που βασίζεται στο λογισμικό.

Η τεκμηρίωση (documentation) της αρχιτεκτονικής είναι εξίσου σημαντική με τη δημιουργία της, είναι ένα σύνολο εγγράφων που αναλύουν τον σχεδιασμό και την συναρμολόγηση ενός συστήματος. Συχνά περιλαμβάνει διαγράμματα αρχιτεκτονικής τα οποία μοιάζουν με χάρτες του συστήματος, απεικονίζουν τα στοιχεία, τις σχέσεις τους και τη ροή δεδομένων, και μπορούν να αναπαραστήσουν διάφορες πλευρές (views) του συστήματος, όπως δομική, συμπεριφορική, και άλλες. Η τεκμηρίωση της αρχιτεκτονικής μπορεί να έχει έναν αριθμό ευεργετικών χρήσεων, όπως η διευκόλυνση της επικοινωνίας όλων των stakeholders για τον σχεδιασμό του συστήματος.

Η γλώσσα ή το σύνολο των συμβόλων και των κανόνων που χρησιμοποιούνται για τη δημιουργία αυτών των αρχιτεκτονικών διαγραμμάτων ονομάζεται αρχιτεκτονική σημειογραφία. Αυτή παρέχει έναν τυποποιημένο τρόπο αναπαράστασης των στοιχείων του συστήματος και των αλληλεπιδράσεών τους. Η UML (Ενοποιημένη Γλώσσα Μοντελοποίησης) έχει γίνει μια ευρέως χρησιμοποιούμενη σημειογραφία που περιλαμβάνει ένα πλήθος από διαγράμματα που βοηθούν τους χρήστες να οπτικοποιήσουν και να τεκμηριώσουν τα συστήματα λογισμικού. Τα διαγράμματα αυτά περιγράφουν διαφορετικές πτυχές του συστήματος και σε διαφορετικό επίπεδο λεπτομέρειας.

Οι αποφάσεις αρχιτεκτονικής λογισμικού βασίζονται συνήθως σε ένα σύνολο από αρχιτεκτονικά στυλ, δηλαδή κάποιες γενικές, επαναχρησιμοποιούμενες λύσεις σε κοινά προβλήματα αρχιτεκτονικής λογισμικού. Βασικά αρχιτεκτονικά στυλ περιλαμβάνουν: τη μονολιθική αρχιτεκτονική - όπου όλη η λειτουργικότητα του συστήματος ενσωματώνεται σε μια εφαρμογή, την Αρχιτεκτονική Προσανατολισμού Υπηρεσιών (SOA) - όπου το σύστημα χωρίζεται σε υπηρεσίες οι οποίες επικοινωνούν μεταξύ τους μέσω ενός Επιχειρησιακού Διαύλου Υπηρεσιών (ESB) και τέλος, την αρχιτεκτονική μικροϋπηρεσιών - όπου το σύστημα αποτελείται από ένα σύνολο χαλαρά συνδεδεμένων, ανεξάρτητων, μικρών υπηρεσιών που επικοινωνούν με ελαφριά πρωτόκολλα (όπως HTTP/REST).

Αρχιτεκτονική Μικροϋπηρεσιών

Η παραδοσιακή μονολιθική αρχιτεκτονική δεν ανταποκρίνεται στις σύγχρονες ανάγκες της κλιμακωσιμότητας και της ταχείας ανάπτυξης, για το λόγο αυτό γίνεται πλέον ευρεία χρήση των μικροϋπηρεσιών.

Η αρχιτεκτονική μικροϋπηρεσιών ευνοεί την ανεξάρτητη ανάπτυξη και αναβάθμιση των υπηρεσιών, μειώνοντας τις επιπτώσεις των αλλαγών σε όλο το σύστημα. Το μοντέλο αυτό επιτρέπει επίσης την ευέλικτη κλιμάκωση, καθώς οι υπηρεσίες μπορούν να κλιμακωθούν ανεξάρτητα μεταξύ τους. Από την άλλη πλευρά, η αρχιτεκτονική μικροϋπηρεσιών μπορεί να επιφέρει αυξημένη πολυπλοκότητα, ειδικά σε ότι αφορά την συντονισμένη διαχείριση και παρακολούθηση των υπηρεσιών. Επίσης, τα ζητήματα ασφαλείας και διαχείρισης δεδομένων μπορεί να είναι πιο πολύπλοκα σε συγκριση με τη μονολιθική αρχιτεκτονική.

Με βάση τα παραπάνω γίνεται εμφανές ότι η αρχιτεκτονική αυτή επηρεάζει χαρακτηριστικά ποιότητας του συστήματος, αφού οδηγεί σε βελτιωμένη διατηρησιμότητα και προσαρμοστικότητα. Ωστόσο, πρέπει να γίνει σωστή διαχείριση της πολυπλοκότητας για να εξασφαλιστεί η αποδοτικότητα, η αξιοπιστία και η ασφάλεια του συστήματος.

Επικοινωνία Μικροϋπηρεσιών

Οι μικροϋπηρεσίες χρειάζεται να συνεργάζονται μεταξύ τους για να ικανοποιήσουν μια επιχειρησιακή ανάγκη. Δύο συνήθη μοτίβα συνεργασίας μικροϋπηρεσιών είναι η Χορογραφία και η Ενορχήστρωση. Στη χορογραφία, οι μικροϋπηρεσίες δημοσιεύουν ένα μήνυμα (event), μέσω ενός διαμεσολαβητή (broker) μηνυμάτων, κάθε φορά που υπάρχει μια αλλαγή στην κατάσταση, και άλλες μικροϋπηρεσίες που εγγράφονται σε αυτό το event ετοιμάζουν τον εαυτό τους για την επόμενη επανάληψη της διαδικασίας. Από την άλλη πλευρά, στην ενορχήστρωση, μια σύνθετη μικροϋπηρεσία λειτουργεί ως ελεγκτής που οργανώνει τη ροή των μηνυμάτων της εφαρμογής, καλώντας πολλαπλές ατομικές υπηρεσίες σε μια σειρά.

Η χορογραφία και η ενορχήστρωση έχουν διαφορετικά πλεονεκτήματα και αδυναμίες. Στη χορογραφία οι μικρουπηρεσίες μπορούν να αναπτυχθούν ανεξάρτητα (ελαφρώς συνδεδεμένη προσέγγιση) και δεν υπάρχει πολύ ΄φλυαρία΄ στα δεδομένα, αλλά η εποπτεία των διαδικασιών είναι δύσκολη και η σχεδίαση των εφαρμογών είναι σχετικά πολύπλοκη. Αντίθετα, η ενορχήστρωση επιτρέπει ευκολότερη παρακολούθηση των διαδικασιών και η σχεδίαση των εφαρμογών είναι σχετικά απλή, αλλά η υπηρεσία ελέγχου μπορεί να μετατραπεί σε ΄κεντρική εξουσία΄ ή να λειτουργήσει ως μοναδικό σημείο αποτυχίας (SPOF).

Η υλοποίηση της χορογραφίας και της ενορχήστρωσης περιλαμβάνει διαφορετικές τεχνολογίες. Για τη χορογραφία, χρησιμοποιούνται brokers όπως το RabbitMQ ή το Redpanda για την επικοινωνία με βάση τα events. Από την άλλη πλευρά, για την ενορχήστρωση, χρησιμοποιούνται τεχνολογίες αιτήματος/απάντησης όπως το RPC και το REST.

Σκοπός και Εργαλεία

Στο σημερινό ταχέως εξελισσόμενο τεχνολογικό τοπίο, πολλοί οργανισμοί αντιμετωπίζουν το πρόβλημα της επιλογής της καταλληλότερης αρχιτεκτονικής για το λογισμικό τους. Το παρόν έργο επιδιώκει να βοηθήσει τους ενδιαφερόμενους (stakeholders), τους αρχιτέκτονες και τις ομάδες ανάπτυξης να λαμβάνουν τεκμηριωμένες αρχιτεκτονικές αποφάσεις για το λογι-

σμικό, διερευνώντας και αξιολογώντας πέντε διαφορετικές αρχιτεκτονικές προσεγγίσεις. Θα αναδείξει τα πλεονεκτήματα και τις αδυναμίες κάθε αρχιτεκτονικής και θα αξιολογήσει την απόδοσή τους χρησιμοποιώντας διάφορες μετρικές, λαμβάνοντας υπόψην τις απαιτήσεις, τους στόχους και τους περιορισμούς του συστήματος.

Αναφερόμενοι στα εργαλεία που χρησιμοποιήθηκαν για την ανάπτυξη των αρχιτεκτονικών μας, η ανάλυσή μας ξεκινάει από τους διαμεσολαβητές μηνυμάτων (brokers). Ένας message broker είναι μια εφαρμογή που λειτουργεί ως ενδιάμεσο στοιχείο για την επικοινωνία μεταξύ διαφόρων εφαρμογών. Οι brokers προσφέρουν δύο βασικά πρότυπα διανομής μηνυμάτων: το μοντέλο point-to-point και το μοντέλο publish/subscribe. Τα RabbitMQ και Redpanda είναι δύο είδη brokers που χρησιμοποιήθηκαν για την υλοποίηση της ασύγχρονης επικοινωνίας.

Το RabbitMQ είναι ένα λογισμικό message broker ανοιχτού κώδικα που υλοποιεί το πρωτόκολλο AMQP. Τα μηνύματα που παράγονται από τον producer μπαίνουν σε μια ουρά (queue) και παραμένουν εκεί έως ότου καταναλωθούν από έναν consumer. Το RabbitMQ λειτουργεί με πολιτική προώθησης (push-based) για την παράδοση των μηνυμάτων στους συνδρομητές, έτσι τα μηνύματα στέλνονται απευθείας στον καταναλωτή χωρίς να χρειαστεί να τα ζητήσει ο ίδιος.

Το Redpanda είναι μια πλατφόρμα ροής μηνυμάτων (events) που παρέχει την υποδομή για ροή δεδομένων σε πραγματικό χρόνο. Οι παραγωγοί μηνυμάτων στέλνουν τα δεδομένα στο Redpanda και αυτό τα αποθηκεύει σειριακά και τα οργανώνει σε θέματα (topics). Οι consumers εγγράφονται σε ένα θέμα ώστε να μπορούν να διαβάσουν τα μηνύματα. Το Redpanda χρησιμοποιεί ένα μοντέλο κατανάλωσης βασισμένο στην ανάκληση (pull-based), που επιτρέπει σε μια εφαρμογή να καταναλώνει δεδομένα με τον δικό της ρυθμό και να ξαναρχίζει την κατανάλωση όποτε είναι απαραίτητο.

Άλλα εργαλεία που χρησιμοποιήθηκαν για την ανάπτυξη αρχιτεκτονικών λογισμικού περιλαμβάνουν το Docker, το Grafana και το Prometheus. Το Docker είναι μια τεχνολογία που επιτρέπει την κατασκευή, την εκτέλεση, τη δοκιμή και την ανάπτυξη διανεμημένων εφαρμογών μέσω εικονικών πακέτων λογισμικού, γνωστών ως containers. Αυτά τα containers προσφέρουν μεγάλη φορητότητα και συνέπεια, καθώς εξασφαλίζουν ότι το λογισμικό θα λειτουργεί πάντα με τον ίδιο τρόπο, ανεξάρτητα από το περιβάλλον του. Σε συνδυασμό με αυτό, το Prometheus είναι ένα σύστημα παρακολούθησης που καταγράφει πραγματικές μετρήσεις σε μια βάση δεδομένων χρονοσειρών, παρέχοντας παράλληλα δυνατότητες ειδοποιήσεων, ενώ το Grafana είναι μια πλατφόρμα παρακολούθησης που επιτρέπει την οπτικοποίηση και την κατανόηση των δεδομένων μετρήσεων.

Τέλος, για την εργασία αυτή χρησιμοποιήθηκαν ακόμα η PostgreSQL, ως βάση δεδομένων της εφαρμογής, η Python ως γλώσσα προγραμματισμού και το Visual Paradigm ως εργαλείο υλοποίησης UML διαγραμμάτων.

Μελέτη Περίπτωσης

Στην παρούσα εργασία αναπτύσσεται μια μελέτη περίπτωσης που εξετάζει την χρήση διαφορετικών αρχιτεκτονικών μικροϋπηρεσιών για την ανάπτυξη ενός μηχανισμού αποθήκης δεδομένων (data warehouse). Στο πλαίσιο αυτό, γίνεται ανάλυση και αποθήκευση δεδομένων κατανάλωσης ενέργειας από το ENTsoe V3 API, για διάφορες χώρες της Ευρωπαϊκής Ένωσης, με σκοπό την εξαγωγή εύχρηστων και εύκολα προσβάσιμων δεδομένων.

Οι λειτουργικές απαιτήσεις περιλαμβάνουν την εξαγωγή δεδομένων από το API, το διαχωρισμό των δεδομένων ανά χώρα και τον υπολογισμό της ημερήσιας, εβδομαδιαίας και μηνιαίας κατανάλωσης για κάθε χώρα. Οι μη λειτουργικές απαιτήσεις περιλαμβάνουν την απόδοση, την κλιμακωσιμότητα και την ακρίβεια του συστήματος.

Τα βασικά στοιχεία (components) των αρχιτεκτονικών μας είναι οι workers, υπεύθυνοι είτε για την αρχικοποίηση της βάσης και του broker (initiator worker), είτε για την εξαγωγή των δεδομένων από το API (data workers), είτε για τον υπολογισμό της ημερήσιας (daily workers), εβδομαδιαίας (weekly workers) και μηνιαίας (monthly workers) κατανάλωσης ενέργειας. Ο μηχανισμός επικοινωνίας μεταξύ των ωορκερς διαφέρει ανάλογα με την αρχιτεκτονική που χρησιμοποιείται και μπορεί να περιλαμβάνει ουρές RMQ, ένα θέμα Redpanda ή websockets.

Αρχιτεκτονικές

Για την παραπάνω μελέτη περίπτωσης υλοποιήθηκαν πέντε διαφορετικές αρχιτεκτονικές. Τα κοινά στοιχεία μεταξύ αυτών είναι:

- Το ENTsoe V3 API παρέχει τα δεδομένα κατανάλωσης ενέργειας για κάθε χώρα.
- Η PostgreSQL βάση δεδομένων όπου αποθηκεύονται η συνολική, η ημερήσια, η εβδομαδιαία και η μηνιαία κατανάλωση.
- Το Prometheus που συνδέεται με τους workers και συλλέγει διάφορες μετρικές του συστήματος.
- Το Grafana που συνδέεται με το Prometheus και χρησιμεύει στην οπτικοποίηση των μετρικών του συστήματος.
- Ολοι οι workers λειτουργούν σε ένα περιβάλλον Docker, σε ξεχωριστά containers.
- Οι data workers εξάγουν τα δεδομένα σε παρτίδες του ενός μήνα (= 1 request).
- Υπάρχει ένας data worker για κάθε χώρα (27 data workers που λειτουργούν ταυτόχρονα).

Συγκεκριμένα, οι αρχιτεκτονικές που υλοποιήθηκαν είναι οι εξής:

1. Orchestrator: Για την επικοινωνία μεταξύ των workers γίνεται χρήση websockets. Ο data worker λειτουργεί ως μια κεντρική μονάδα ελέγχου (ενορχηστρωτής). Αφού στείλει τα συνολικά δεδομένα στον daily, στέλνει (τα ημερήσια δεδομένα) στον weekly και τέλος (τα εβδομαδιαία δεδομένα) στον monthly worker.
2. Serialised Orchestrator: Επίσης χρησιμοποιεί websockets για την επικοινωνία, αλλά η επικοινωνία μεταξύ των workers γίνεται σειριακά. Ο data worker στέλνει στον daily, ο daily στον weekly και ο weekly στον monthly.
3. Async Orchestrator: Χρησιμοποιεί επίσης websockets, αλλά ο data worker στέλνει το συνολικό φορτίο σε όλους τους daily, weekly και monthly workers ταυτόχρονα.

4. Serialised RabbitMQ: Σε αυτήν την αρχιτεκτονική η επικοινωνία πραγματοποιείται με τη χρήση του διαμεσολαβητή μηνυμάτων RabbitMQ. Τα μηνύματα γράφονται σε τρεις ουρές μηνυμάτων, τις daily, weekly και monthly queues.

5. SOA Redpanda: Η επικοινωνία πραγματοποιείται με τη χρήση του διαμεσολαβητή μηνυμάτων Redpanda. Ο data worker γράφει σε ένα θέμα (topic) και οι daily, weekly και monthly workers εγγράφονται σε αυτό το θέμα ώστε να διαβάζουν τα μηνύματα.

Το deployment των αρχιτεκτονικών έγινε σε 4 servers, ο καθένας με διακριτούς ρόλους και προδιαγραφές. Αναλυτικά οι servers: ο Front server περιλαμβάνει τα Metabase και Grafana, ο Broker Server περιλαμβάνει τον broker (εφόσον υπάρχει), ο DB Server για την βάση δεδομένων και ο Workers Server για τους workers και το Prometheus.

Επιπλέον, έχουν δημιουργήσει αρκετά UML διαγράμματα, συγκεκριμένα διαγράμματα στοιχείων (component), διαγράμματα ακολουθίας (sequence) και διαγράμματα ανάπτυξης (deployment), για να παρέχουμε μια αναλυτική οπτική αναπαράσταση της κάθε αρχιτεκτονικής.

Συγκριτική Αξιολόγηση και Αποτελέσματα

Για την αξιολόγηση των αρχιτεκτονικών που υλοποιήθηκαν, παρουσιάζουμε διάφορες μετρικές και διεξάγουμε πλήθος δοκιμών (tests). Οι δοκιμές συστήματος αποτελούν τη βάση οποιασδήποτε αξιόπιστης στρατηγικής ανάπτυξης λογισμικού, προσφέροντας ζωτικές πληροφορίες για την απόδοση, την επεκτασιμότητα και τη σταθερότητα του συστήματος.

Ο αριθμός των data workers έμεινε σταθερός καθ'όλη τη διάρκεια των δοκιμών (27 data workers). Οι δοκιμές ξεκίνησαν με την εκτέλεση του συστήματος με έναν worker από κάθε είδος (ημερήσιο, εβδομαδιαίο, μηνιαίο), ενώ εξομοιώθηκε η επεξεργασία δεδομένων 1 έτους, 3 ετών και 6 ετών. Συνεχίσαμε αυξάνοντας τον αριθμό των workers σε τρεις για κάθε είδος και διεξήγαμε τις ίδιες δοκιμές.

Αξίζει να σημειωθεί ότι όλοι οι data workers συντονίστηκαν για να ξεκινήσουν ταυτόχρονα. Επίσης, στην αρχή κάθε δοκιμής, η βάση δεδομένων ήταν κενή και όλοι οι πίνακες διαγράφονταν μετά από κάθε δοκιμή και δημιουργούνταν ξανά στην επόμενη εκτέλεση. Τα στοιχεία αυτά ήταν αναγκαία για την εξασφάλιση ίδιων συνθηκών μεταξύ των δοκιμών.

Κατά τη διάρκεια των δοκιμών αντιμετωπίστηκαν κάποια προβλήματα, συμπεριλαμβανομένης της σταδιακής εξάντλησης χώρου δίσκου από το Docker (λόγω κάποιας διαρροής) και της υψηλής κατανάλωσης εύρους ζώνης δικτύου από το Prometheus. Επίσης, μέσω των δοκιμών και των αποτελεσμάτων, μας έγιναν γνωστά κάποια σφάλματα στην υλοποίηση του συστήματος τα οποία, βέβαια, διορθώθηκαν πριν γίνουν οι τελικές μετρήσεις.

Με βάση τις δοκιμές παρατηρήθηκαν διάφορα σημαντικά αποτελέσματα και εξήχθησαν κάποια συμπεράσματα.

Μερικές αρχικές μετρήσεις είναι οι εξής: ο μέσος χρόνος ανά αίτημα στο ENTsoe V3 ήταν 1 δευτερόλεπτο, ο συνολικός αριθμός requests για 1 έτος ήταν 405, για 3 έτη ήταν 1053 και για 6 έτη ήταν 2025.

Η βασική μετρική που αξιολογήθηκε ήταν ο μέσος χρόνος που απαιτείται από τη στιγμή που ο data worker έχει λάβει την απάντηση ενός request στο API, μέχρι τη στιγμή που τα επεξεργασμένα δεδομένα αποθηκεύονται στη βάση δεδομένων, από έναν άλλο worker (ημερήσιο, εβδομαδιαίο ή μηνιαίο). Όσον αφορά αυτή τη μετρική, διαπιστώθηκε ότι σε κάποιες

αρχιτεκτονικές, ο χρόνος μειώνεται καθώς ο όγκος των δεδομένων αυξάνεται (δηλαδή όσο αυξάνονται τα έτη, από ένα έως έξι). Η Redpanda αποδείχθηκε η πιο αργή αρχιτεκτονική, ενώ η αρχιτεκτονική Orchestrator αποδείχθηκε η πιο γρήγορη. Ακόμη, μεταξύ των διαφόρων Orchestrator αρχιτεκτονικών, η Async Orchestrator ηταν η πιο αργή.

Όταν οι δοκιμές διεξήχθησαν με 3 εργάτες κάθε τύπου (καθημερινά, εβδομαδιαία, μηνιαία) να λειτουργούν παράλληλα, διαπιστώθηκε μια σημαντική βελτίωση στον μέσο χρόνο από το request σε κάθε worker. Η παράλληλη λειτουργία των workers βελτίωσε δραματικά την επίδοση όλων, εκτός της αρχιτεκτονικής Redpanda. Επίσης, οι διάφορες αρχιτεκτονικές εμφάνισαν παρόμοιους χρόνους, γεγονός που πιθανά οφείλεται σε συμφόρηση λόγω της φύσης της σύνδεσης (TCP). Ακόμη, η Serialised RabbitMQ παρουσίασε παρόμοιο χρόνο με τις Orchestrator αρχιτεκτονικές αφού είναι και οι δύο push-based.

Για την επόμενη μετρική, τον συνολικό χρόνο εκτέλεσης, δεν υπήρχαν σημαντικές διαφορές μεταξύ των αρχιτεκτονικών, αφού όλες ολοκλήρωσαν τα requests στον ίδιο περίπου χρόνο (89s για 6 έτη δεδομένων). Αυτό δηλώνει την εγκυρότητα των μετρικών, αφού εάν κάποια αρχιτεκτονική ήταν πιο αργή τότε θα είχε και διαφορετικό CPU time, δηλαδή περισσότερους διαθέσιμους πόρους.

Για να δοκιμαστεί το σύστημα υπό μη κανονικές συνθήκες, τα δεδομένα του API αποθηκεύτηκαν τοπικά, ώστε να γίνει ταυτόχρονη εκτέλεση όλων των requests. Σε αυτή την περίπτωση παρατηρήθηκε ένα νέο σημείο συμφόρησης (στα περίπου 62s) αφού το σύστημα έφτασε το όριο χρήσης CPU.

Ο πηγαίος κώδικας για την εφαρμογή που αναπτύχθηκε σε αυτήν τη διατριβή παρέχεται στο Παράρτημα Β για περαιτέρω αναθεώρηση και ανάλυση. Τα αποτελέσματα των δοκιμών είναι επίσης διαθέσιμα σε μορφή snapshots από το Grafana, στο Παράρτημα Α.

Συμπεράσματα

Σκοπός της εργασίας είναι η παρουσίαση των αποτελεσμάτων των δοκιμών και όχι η εξαγωγή συμπερασμάτων για την σύγκριση των αρχιτεκτονικών. Βέβαια, μέσα από τις δοκιμές που διεξήχθησαν και με στόχο την αξιολόγηση των αρχιτεκτονικών μας, παραθέτουμε μερικά συμπεράσματα.

- Η SOA Redpanda είναι ιδανική για συστήματα με πολλούς παραγωγούς/καταναλωτές που πρέπει να είναι stateless, δηλαδή να μην υπάρχει κάποιος καταναλωτής που να είναι υπεύθυνος για την ανθεκτικότητα (resiliency) της αρχιτεκτονικής. Επίσης, η Redpanda επιτρέπει την επανάληψη δεδομένων και την άμεση κλιμάκωση. Αντίθετα, για συστήματα όπου η καθυστέρηση ανά αίτημα είναι πρωταρχικής σημασίας, η Redpanda μπορεί να μην είναι η βέλτιστη επιλογή.

- Η Serialised RabbitMQ είναι υψηλής απόδοσης και αρκετά ευέλικτη, ιδανική για περιπτώσεις όπου η προτεραιότητα κατανάλωσης δεδομένων (με χρήση queue) είναι ζωτικής σημασίας. Όμως, μπορεί να μην είναι η καλύτερη επιλογή όταν η προτεραιότητα κατανάλωσης δεν είναι τόσο σημαντική.

- Ο Orchestrator είναι ο ιδανικός όταν χρειάζεται ένα κεντρικό σημείο ελέγχου (π.χ. σε φιλτράρισμα δεδομένων) για τη διαχείριση ανεξάρτητων υπηρεσιών. Όμως, μπορεί να

μην είναι ιδανικός για συστήματα που πρέπει να είναι stateless.

- Ο Async Orchestrator είναι εξαιρετικός για συστήματα με πλεόνασμα CPU, αλλά μπορεί να δημιουργήσει προβλήματα εάν οι πόροι της CPU είναι περιορισμένοι.
- Τέλος, ο Serialized Orchestrator είναι χρήσιμος όταν η επεξεργασία μεταδεδομένων προσθέτει σημαντική αξία στην απόδοση (π.χ. λόγω μείωσης όγκου πληροφορίας).

Το σύστημα που αναπτύχθηκε στα πλαίσια αυτής της διπλωματικής εργασίας θα μπορούσε να βελτιωθεί και να επεκταθεί περαιτέρω. Συγκεκριμένα, η εξερεύνηση εναλλακτικών στοιβών λογισμικού, όπως Golang ή Rust, μπορεί να οδηγήσει σε διαφορετικά αποτελέσματα. Επιπλέον, μια πιο εμπεριστατωμένη αξιολόγηση των συμπερασμάτων μας μέσω αυστηρών διαδικασιών επικύρωσης θα μπορούσε να συμβάλει σημαντικά στη διεύρυνση της έρευνας. Τέλος, η χρήση συνδυασμού διαφόρων μεθόδων επικοινωνίας μπορεί να επηρεάσει διαφορετικά την απόδοση και την αξιοπιστία του συστήματος.

Contents

Chapter 1

Software Architecture

The domain of software architecture has grown to be a significant aspect of software engineering, particularly in managing the development and maintenance of large-scale systems. Practitioners have come to realize that getting an architecture right is a critical success factor for system design and development. [1] There are many definitions of software architecture [2], [3], among which is the following: "The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them and properties of both."

The core of all the definitions is the same: the architecture of a system describes its high-level structure. This structure reveals things such as how the system is composed of interacting parts, where are the main pathways of interaction, and what are the key properties of the parts. In other words, the architecture is presented as a description of a system as a sum of smaller parts, and shows how those parts relate to and cooperate with each other to perform the work of the system. [4] Software architecture typically plays a key role as a bridge between requirements and implementation. By providing an abstract description of a system, the architecture exposes certain properties, while hiding others. This representation provides a guide to the overall system, permits designers to reason about the ability of a system to satisfy certain requirements, and suggests a blueprint for system construction and composition. [3], [5]

Software architecture is planned out during the initial stages of the development process. It either enables or restricts the realization of particular functional needs, non-functional requirements and business goals. As a result, making the right architectural choices is fundamental to the success of a software-based project. [1]

1.1 Documenting software architecture

As we have already seen, software architecture plays a central role for the successful development and maintenance of large complex systems. The architecture serves as a blueprint and defines the work assignments that must be carried out by the various teams (e.g. design and development). Moreover, quality attributes desired for a system depend heavily on the architecture, such as performance, accuracy, reliability, scalability and security.

Documenting the architecture is as important as crafting it. Even a perfect architecture

is useless if no one understands it or (perhaps even worse) if key stakeholders misunderstand it. [2] Architecture documentation can have a number of beneficial uses, such as [4]:

- Facilitating communication about the system's design with relevant stakeholders throughout its evolution.
- Providing a foundation for initial analysis to verify the soundness of architectural design decisions, identify potential weaknesses, and make necessary refinements or changes.
- Serving as the initial resource for gaining comprehensive understanding of the system.

So, the problem here is "How should you document an architecture so that others can successfully use it, maintain it, and build a system from it?" The answer given by Clements et al. [6] is an approach called "views and beyond". [7]

Views

The basic concept associated with software architecture documentation is the concept of architectural views. A view is a representation of a set of system elements and relations associated with them. Due to their complexity, comprehending modern systems can be challenging. Therefore, we typically focus on one, or a small set of, software system structures at a time, which we represent as views.

The first principle for documenting software architectures is to document the relevant views and then document the information that applies beyond views. However, there are different approaches to finding out which views are relevant, Rational's 4+1 approach prescribes one set; the Siemens Four Views approach prescribes another set. A recent trend, however, is to recognize that architects should produce whatever views are useful for the system at hand. IEEE 1471 holds that an architecture description consists of a set of views, each of which conforms to a viewpoint, which in turn is a realization of the concerns of one or more stakeholders.

Views can be usefully grouped into viewtypes, corresponding to the three broad ways an architect must think about a system: as a set of implementation units - module viewtype, as a set of runtime elements interacting to carry out the system's work - component-and-connector viewtype, and as a set of elements existing in and relating to external structures in its environment - allocation viewtype. Within each viewtype, there are well-known patterns of design decisions (styles) that the architect can re-use. [6]

Diagrams

Architectural documentation refers to the collection of documents that communicate the design and describe how a system is put together. It generally includes architectural diagrams (often created using architectural notations), as well as descriptions of architectural decisions, system components, interaction mechanisms, constraints, justifications, and more.

An architectural diagram is a visual representation of a system's architecture. It's like a map, depicting components of the system, their relationships, interactions, and flow

of data. The diagrams could represent various views of the system, like structural view, behavioral view, or deployment view. These diagrams are very useful for understanding, communicating, and documenting the system's architecture.

The language or the set of symbols and rules used to create these architectural diagrams, is called an architectural notation. These notations provide a standardized way to represent system components and their interactions, making the diagrams easier to understand and share among different stakeholders. Examples of architectural notations include Unified Modeling Language (UML), Systems Modeling Language (SysML), and Architecture Description Language (ADL).

UML

Since we have used a variety of UML diagrams in order to describe our architectures, we are going to talk a little bit more about Unified Modeling Language. Although informal box-and-line sketches communicated on viewgraphs may still be the most popular form of architectural expression, the UML has become a widely used notation.

UML, short for Unified Modeling Language, is a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML is a very important part of developing object oriented software and the software development process. The UML uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software. [8]

The first thing to notice about the UML is that there are a lot of different diagrams (models) to get used to. The reason for this is that it is possible to look at a system from many different viewpoints. A software development will have many stakeholders playing a part, all of which are interested in different aspects of the system, and each of them require a different level of detail.

UML offers two main types of diagrams [8]:

- Structure diagrams, which show the static structure of the system and its parts on different abstraction and implementation levels and how they are related to each other. These include class diagrams, component diagrams, deployment diagrams etc.
- Behavior diagrams, which show the dynamic behavior of the objects in a system, which can be described as a series of changes to the system over time. These include use case diagrams, activity diagrams, sequence diagrams, communication diagrams etc.

From the above-mentioned types, we are going to explain only the ones we will use in our project.

- Component diagram: it depicts how components are wired together to form larger components or software systems. It illustrates the architectures of the software components and the dependencies between them.

- Deployment diagram: it shows the architecture of the system as deployment (distribution) of software artifacts to deployment targets. Artifacts represent concrete elements in the physical world that are the result of a development process. It models the run-time configuration in a static view and visualizes the distribution of artifacts in an application. In most cases, it involves modeling the hardware configurations together with the software components that lived on.

- Sequence diagram: it models the collaboration of objects based on a time sequence. It shows how the objects interact with others in a particular scenario of a use case.

- Communication diagram: it is also used to model the dynamic behavior of the use case. When compared to Sequence Diagram, the Communication Diagram is more focused on showing the collaboration of objects rather than the time sequence.

1.2 Architectural styles

Software architecture decisions can rely on a set of idiomatic patterns commonly named architectural styles or patterns. A software architectural pattern defines a family of systems in terms of a pattern of structural organization and behavior. [9] In other words, architectural patterns are a general, reusable solution to a commonly occurring problem in software architecture within a particular context. They are best practices that the software development community has found to work well in some situations. Some common architectural patterns include Monolithic, Service Oriented Architecture (SOA), Publisher-Subscriber, Microservices, Model View Controller (MVC), Layered, Multi-tier, Event-driven, Pipe-Filter and Client–Server.

System-wide architectural styles

Monolithic, SOA and Microservices architectures are all high-level architectural patterns that dictate the overall structure and organization of a software system. In the next chapters, we are going to stir our attention towards the Microservices architecture and explain the concepts of orchestration and choreography in detail. But before we proceed, it's worthwhile to briefly examine the historical context and interplay among these three key architectures. This historical backdrop will provide a richer understanding and pave the way for our forthcoming detailed discussion.

1.2.1 Monolithic

Monolithic architecture is a traditional method of software development, which has been used by large companies such as Amazon and eBay in the past. [10] In a monolithic architecture, all functionality is encapsulated into one single application, its modules rely on the sharing of resources of the same machine (memory, databases, or files), so they cannot be executed independently. [11] This type of architecture is tightly-coupled, and all your logic for handling a request runs in a single process.

While it is a good idea to start a project using this type of architecture, because this allows you to explore both the complexity of a system and its component boundaries, once the application becomes large and the team grows in size, this architecture also has some drawbacks. A notable problem of monoliths involves scalability and, in general, all the aspects related to change, since a change made to a small part of the application requires the entire monolith to be rebuilt and deployed. [12]

1.2.2 Service Oriented Architecture (SOA)

In the 1990s, SOA was proposed as a revolutionary innovation to decouple service-side applications and improve the reuse of components. The SOA architecture could be divided into multiple server application oriented functions of loosely coupled services, each service can be managed in different containers, between services through an enterprise service bus to communicate, and share the same database. [10]

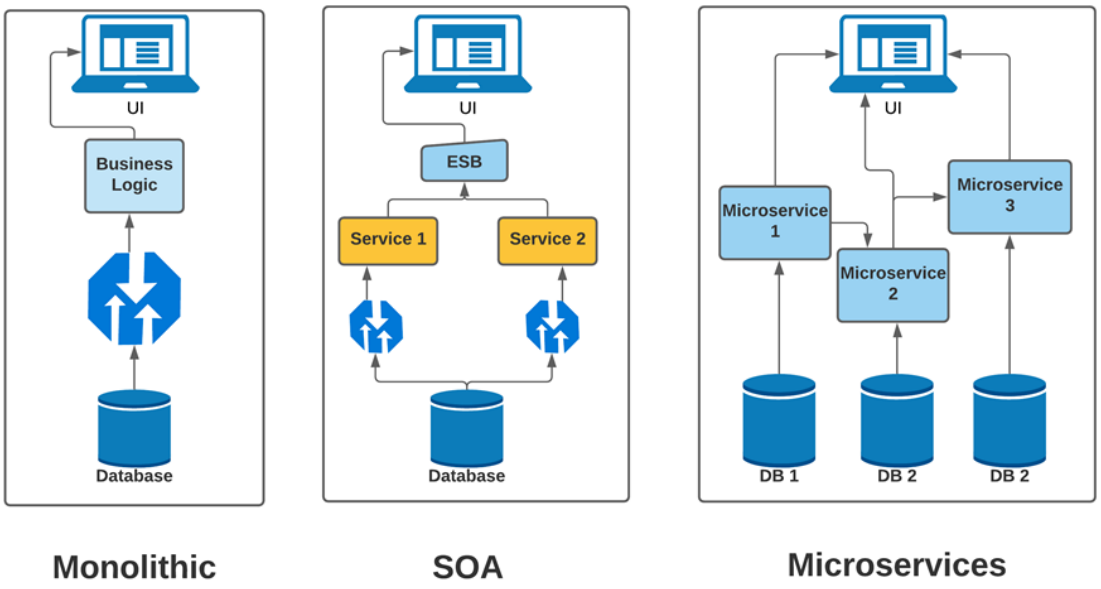


Figure 1.1: *Diagram of Monolithic, Service-Oriented and Microservices Architectures [13]*

1.2.3 Microservices

The microservices architecture style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms. Microservices are built around business capabilities and independently deployable by fully automated deployment machinery. Because of their size, they are easier to maintain and more fault-tolerant since the failure of one service will not break the whole system, which could happen with a monolithic architecture. [12]

Both microservices architecture and SOA are considered service-based architectures, meaning that they are architecture patterns that place a heavy emphasis on services as the primary architecture component used to implement and perform business and nonbusiness

functionality. [14] However, SOA and microservices have a number of differences. First, the size of a Microservice is much smaller than that of an SOA. Also, SOA looks to reuse components, while microservices are all about minimizing the reuse of code. Thirdly, SOA services obtain data from a central location, while microservices use a local data source for each service. [13]

The market's high pace of demand for new application features requires changes both in the applications themselves (loose coupling and high scalability) and in the way they are built (loose team dependencies and fast deployment). Microservices address both concerns since small services can be built and deployed by independent development teams; the concomitant freedom allows teams to focus on improving each service and increase business value. [12]

It is worth mentioning that many companies, such as Walmart, Spotify, Netflix, Amazon, and eBay, among others, and many other large-scale websites and applications have evolved from a monolithic architecture to microservices. [15] Moreover, a significant body of contemporary literature engages with case studies and examines methodologies for migrating from monolithic applications (built of interconnected, interdependent components) to collections of large services, like SOA and further to collections of small, autonomous, lightweight-connected services: the microservices architecture. [11], [12], [16]

Chapter 2

Microservices Architecture

Currently, organizations face the need to create scalable applications in an agile way that impacts new forms of production and business organization. The traditional monolithic architecture no longer meets the needs of scalability and rapid development. This is why companies must adopt new technologies and business strategies. [15]

Definition

Microservices is a modern approach to software development that structures an application as a collection of loosely coupled, independently deployable sets of small services which are often developed, deployed, and maintained by a single team. Each service is running in its own process and communicating with other services by using lightweight mechanisms (e.g. HTTP/REST). [12], [17]

Key features of a good service

Each service has to be in sync with two key concepts: loose coupling and high cohesion. When services are loosely coupled, a change to one service should not require a change to another. Moreover, a loosely coupled service knows as little as it needs to about the services with which it collaborates. High cohesion means that we want related behavior to sit together, and unrelated behavior to sit elsewhere. If we want to change behavior, we want to be able to change it in one place, and release that change as soon as possible. [17], [18]

2.1 Advantages of Microservices

The advantages of microservices are numerous and diverse. While many of these advantages can be attributed to any distributed system, it's the extent to which microservices apply the principles of distributed systems and service-oriented architecture that they often realize these benefits more substantially. The main benefits can be reviewed in the following key points [19]:

- Technology Heterogeneity: Since the system is composed of multiple collaborating services, each one can use different technology.
- Resilience: In a monolithic service, if the service fails, everything stops working, but

in microservices, a failure does not cascade, the problem can be isolated and the rest of the system can keep on working.

- Scaling: Not all parts of the system have to scale together. Microservices allow for scaling in some systems while the rest can work in smaller, less powerful hardware.
- Ease of deployment: A change can be made to a single service and be deployed independently of the rest of the system, which allows for faster code deployment.
- Composability: A functionality can be consumed (reused) in many different ways and for different purposes.
- Organizational Alignment: The team can be distributed, which usually results in higher productivity.

Figure 2.1: *The advantages of Microservices [20]*

Microservice architecture (MSA) has undoubtedly become the most popular modern-day architecture, often used in conjunction with the rapidly advancing public cloud platforms to reap the best benefits of scalability, elasticity and agility. [21] Though MSA is highly advantageous and comes with a huge set of benefits, it has its own set of challenges.

2.2 Challenges of Microservices

MSA introduces several challenges due to its inherent complexity and the distribution of services. Maintaining consistency, monitoring, alarming and fault tolerance are difficult for a distributed system, which means that you have to operate a much more complex system than in monolithic architectures. [22] Based on various literature sources, a number of challenges can be identified [10], [23], [24]:

- One challenge lies in skill and knowledge gaps, especially at higher management levels, where there can be difficulty in understanding microservices. This is because they are usually more complex to set up and maintain. There are also challenges in finding the right technical talent and shifting the mindset of those accustomed to traditional monolithic architectures.
- Identifying suitable service boundaries, commonly referred to as service discovery. Getting service boundaries wrong can be costly, so we should keep related code

together and attempt to reduce the coupling to other services in the system. That is why decomposing, the process of breaking down a large monolithic application into smaller, individual services, is a widely discussed issue with various approaches (such as Single Responsibility Principle or Domain-Driven Design (DDD)).

- Maintaining data consistency across different services can be difficult, as each service typically has its own database. In a monolithic system, data is stored centrally in the same database; while in a microservices architecture, data is distributed in databases of different microservices, so ensuring consistency across multiple databases is a critical issue.
- Network reliability, latency, and bandwidth become more critical and complex to manage with a microservices architecture. Network communication, also, negatively affects the performance of microservices.
- With multiple services interacting, it becomes challenging to understand, test, and verify the system's behavior under normal or abnormal conditions.
- Microservices systems are essentially concurrent distributed systems. In general, the effective way to debug concurrent distributed systems is to track and visualize the execution of the system. The current debugging of microservices systems depends largely on the developer's experience with the system and similar failure cases, and mainly relies on manual methods to check logs.

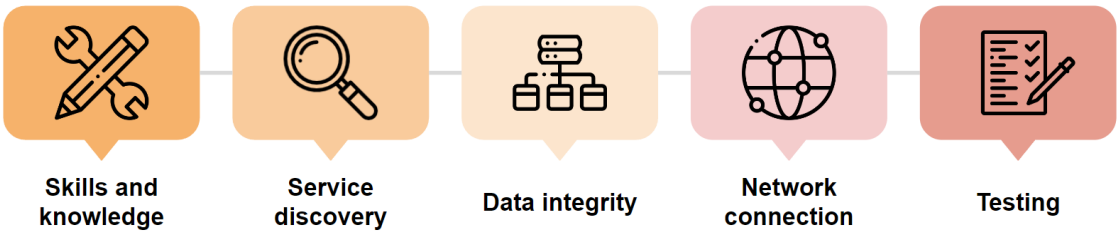


Figure 2.2: *The challenges of Microservices*

Despite these challenges, many still recommend MSA, especially for distributed teams that can logically separate their operations.

2.3 Quality attributes and Microservices

In this section, we are going to look into how microservices affect some of the basic quality attributes. These are the characteristics or properties of a system that can be used to determine its level of quality. They are usually non-functional requirements that describe how the system works.

Availability: Availability is a major concern in microservices as it directly affects the success of a system. Even if a single service is not available to satisfy a request, the whole system may be compromised and experience direct consequences. It has been found that

the size of a microservice is related to its fault proneness. However, it is possible to keep optimal size for services, which may theoretically increase availability. Spawning an increasing number of services, as a system grows larger, will make the system fault-prone on the integration level, which will result in decreased availability due to the large complexity associated with making dozens of services instantly available. [25]

Performance: The prominent factor that negatively impacts performance in the microservice architecture is communication over a network. The network latency is much greater than that of memory. In other words, in-memory calls (i.e. procedure or function calls that run in the same memory space - they are typical in a monolithic architecture) are much faster to complete than sending messages over the network. [25] Therefore, the application must minimize the number of messages sent between each service to prevent the time behavior of the system from being severely affected. This can be accomplished by reducing the responsibility of each microservice and avoiding tight coupling to ensure that communication between services is restricted to the lowest amount. [26]

Scalability: Scalability is the potential to implement more advanced features of the application. Each microservice can be independently scaled leading to improved resource utilization. System scalability is increased, compared to monolithic, because of the smaller services. Scaling may be either vertical, or horizontal.

Reliability: Particular attention should be paid to the reliability of message-passing mechanisms between services and to the reliability of the services themselves. Simple components with clean interfaces can enhance reliability. Integration, a key aspect of microservices, can pose a risk to reliability, especially when the fallacy of assuming network reliability is considered. This factor makes microservices potentially less reliable than applications with in-memory calls, a challenge common to all distributed systems. Microservices streamline integration mechanisms by eliminating additional, potentially complex features and concentrating exclusively on reliable message delivery. [25]

Security and Data integrity: With microservices, security becomes a challenge primarily because no middleware component handles security-based functionality. Instead, each service must handle security on its own, or in some cases the API layer can be made more intelligent to handle the security aspects of the application. [14] Also, choosing to use the microservice architecture will most likely negatively affect the security of the system due to the communication being over a network. Since the messaging mechanism is what allows the different services to communicate with each other, security becomes a necessity to implement for the messaging to ensure high confidentiality and integrity. Confidentiality is of the utmost importance to guarantee that the data sent can only be accessed by authenticated parties. Maintaining high levels of data integrity, through methods like encryption, is crucial to ensure the protection of all transmitted data. [26]

Chapter 3

Microservices Communication

In a microservices-based architecture, an application is composed of multiple microservices, each executing a part of the required functionality. Microservices need to cooperate and collaborate in order to satisfy a business need. Rather than language-level method calls as in a monolith, microservices communicate via standards-based protocols, e.g. HTTP for invocation-based communication, or message-based communication via a message broker. In this note, the microservice collaboration patterns which are commonly used are Orchestration and Choreography. [27]

3.1 Choreography

In this pattern, there is no 'controller' of the end-to-end application process flow. Instead, microservices will publish an event (message), via a message broker, whenever there is a state change, whereby other microservices subscribing to that event then set themselves up for the next iteration of the process. [27]

With choreography, we inform each part of the system of its job, and let it work out the details, like dancers all finding their way and reacting to others around them in a ballet.

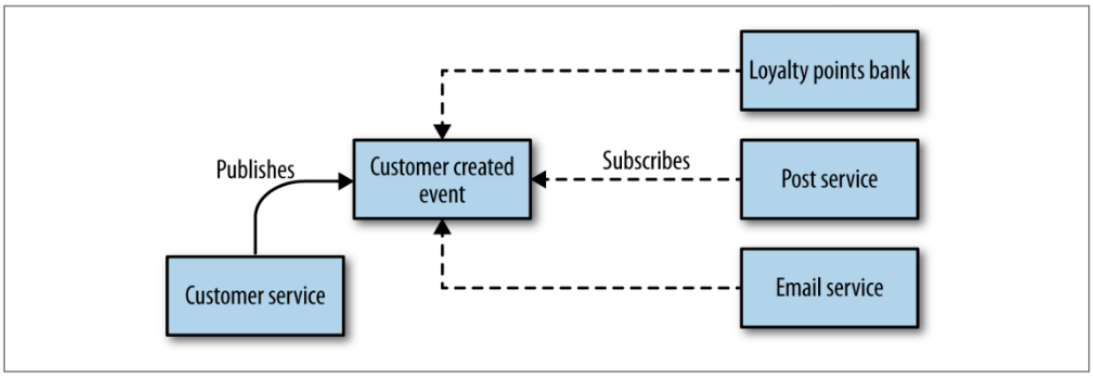


Figure 3.1: *Example of customer creation via choreography [19]*

Strengths:

- It is a loosely coupled approach, meaning that microservices can be deployed independently.

- There is low chattiness, in other words, data is exchanged between microservices only if there is a state change.

Weaknesses:

- End-to-end processes are difficult to monitor (poor visibility). Also, point-to-point connections can lead to 'spaghetti' architectures which are inherently unmanageable. This pattern is less suitable for larger applications where the number of microservices rises.
- Due to the poor visibility of end-to-end processes, and the need for a message broker to intermediate the process flow, the design of applications becomes relatively complex.
- If a microservice goes offline temporarily during a process iteration, the process will eventually complete when the microservice goes back online, so the response time is indeterminate.

3.2 Orchestration

Here, a composite microservice acts as the 'controller' which orchestrates the end-to-end application process flow by invoking multiple atomic services in a sequence. Microservice invocation is done via request/reply interaction, during the process execution (instead of event-based publish-subscribe interaction). [27] With orchestration, we rely on a central brain to guide and drive the process, much like the conductor in an orchestra.

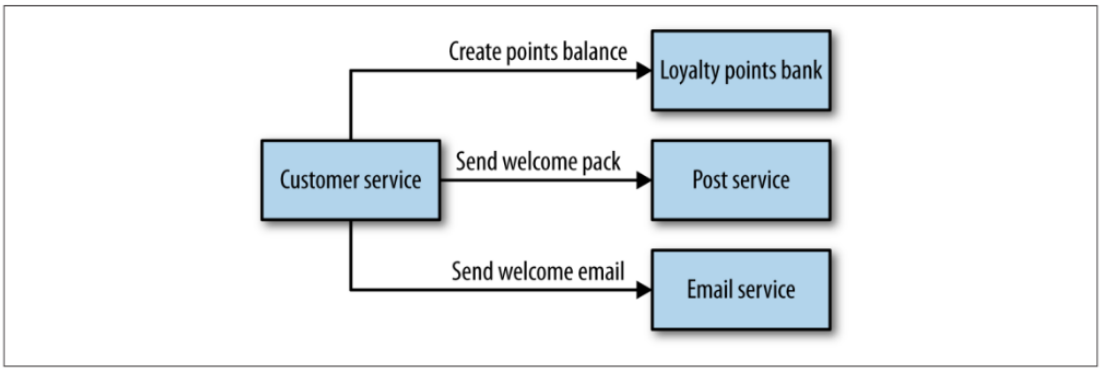


Figure 3.2: *Example of customer creation via orchestration [19]*

Strengths:

- End-to-end processes are easy to monitor (assuming we use synchronous request/response, we could even know if each stage has worked).
- Due to the clear visibility of end-to-end processes, and point-to-point style of invocation-based communication, the design of applications is relatively simple.
- Each step in the application process flow uses invocation-based request-reply interaction, so the response time is predictable.

Weaknesses:

- The 'controller' service can become too much of a central governing authority.
- The orchestrator service can act as a single point of failure, so if it fails, it will stop the entire system from working.
- It is a tightly coupled approach, microservices can be deployed independently, but require downtime during deployment in order to avoid interruption of the application process flow.
- There is high chattiness, which means that data is exchanged between microservices during each step of the application process flow.

According to Newman [19], systems that tend more towards the choreographed approach are more loosely coupled, and are more flexible and amenable to change. Also, in [14] by Richards, it is stated that microservices architecture favors service choreography over service orchestration, primarily because the architecture topology lacks a centralized middleware component.

3.3 Implementing choreography

In Choreography, event-based, asynchronous communication is used. Here there are two things we need to consider: a way for our microservices to emit events, and a way for our consumers to find out those events have happened. [19]

One strategy involves using HTTP to broadcast events. ATOM, which is a REST-compliant specification, provides a means of publishing resource feeds. Consequently, a service, such as customer service, can post an event to an ATOM feed when any changes occur. Consumers then simply poll the feed for any updates. However, this approach has its drawbacks: HTTP doesn't excel in low latency as some specialized message brokers do, and consumers are left with the responsibility of tracking viewed messages and controlling their own polling schedules.

This brings us to the more traditionally used approach, message brokers. Message brokers like RabbitMQ or Redpanda try to handle both problems of asynchronous communication. Producers use an API to publish an event to the broker. The broker handles subscriptions, allowing consumers to be informed when an event arrives. These brokers can even handle the state of consumers, for example by helping keep track of what messages they have seen before. These systems are normally designed to be scalable and resilient. As a consequence, though, these systems can introduce additional complexity and overhead, as it is another system you may need to run to develop and test your services.

Since both RabbitMQ and Redpanda are going to be used in our architectures, we will take a better look at them in chapter 4, where we talk about the message brokers and tools used in our project.

3.4 Implementing orchestration

In Orchestration, we use synchronous request/response. Two technologies fit well when we are considering request/response: remote procedure call (RPC) and REpresentational State Transfer (REST). [19]

- RPC: A type of protocol that allows a program on one computer to execute a program on a server computer. In the context of HTTP, the client makes an HTTP request that represents a method or function call to the server. The server executes the function and returns the result as an HTTP response.
- REST: An architectural style that defines a set of constraints to be used when creating web services. It uses standard HTTP methods, such as GET, POST, PUT, DELETE, etc., to perform operations. In a RESTful system, resources are identified by their URLs, and are accessed and manipulated using HTTP protocol methods. For example, if you wanted to retrieve (or "GET") a particular resource, you could do so by sending an HTTP GET request to the resource's URL.

Both RPC and REST use HTTP as their underlying protocol for communication. While this protocol can be suited well to large volumes of traffic, it isn't great for low-latency communications when compared to alternative protocols that are built on top of Transmission Control Protocol (TCP) or other networking technology.

This is where alternatives like WebSockets can be advantageous. WebSockets operate distinctly from the standard web protocols. Following an initial HTTP handshake, WebSockets essentially form a TCP connection between client and server, making it a more efficient tool for data streaming, particularly to a browser. However, it's worth noting that in this mode of operation, you're barely using HTTP, and REST principles do not apply.

In general, the following need to be kept in mind. Synchronous calls are simpler, and we get to know if things worked straight away. If we like the semantics of request/response but are dealing with longer-lived processes, we could just initiate asynchronous requests and wait for callbacks. On the other hand, asynchronous event collaboration helps us adopt a choreographed approach, which can yield significantly more decoupled services—something we want to strive for to ensure our services are independently releasable. [19]

Chapter 4

Motivation and Tools

In this chapter, we are first going to introduce the problem that motivated us to write this thesis. Subsequently, we will present quantitative measurements and fundamental tools that are going to be used in our project, which is a prerequisite for fully understanding the architectures described afterwards.

4.1 Motivation and objective

In today's rapidly evolving technological landscape, many organizations face the problem of selecting the most suitable architecture for their software. The choice of architecture is crucial for the system's scalability, maintainability, performance and overall success. For the right choice to be made, the system's requirements, goals and constraints need to be taken into account. Since there are numerous architectures available, including microservices, service-oriented architecture (SOA), monolithic, event-driven and more, making a decision can be complex and challenging.

This project aims at helping out stakeholders, architects and development teams by providing the knowledge necessary to make an informed architectural decision. The objective is to explore and evaluate five different architectural approaches for a case scenario, showing the strengths and weaknesses of each architecture, while also observing their performance by calculating different metrics.

4.2 Quantitative measurements

At this point we are going to refer to the quantitative measurements that are a key to evaluating the performance of architectures in a business environment. These measurements provide concrete data that can be used to compare different systems and make informed decisions about which architecture is most suitable for a specific application or workload. The most commonly used quantitative measurements for evaluating architectures include performance metrics, scalability metrics, reliability metrics, and cost metrics. [28]

- Performance metrics are essential indicators of how well a system is operating. For example:

  - Latency: The time it takes for a system to respond to a request. Lower latency often means better performance.
  - Throughput: The amount of data processed by a system in a given amount of time. Higher throughput indicates the system can handle larger amounts of data.
  - Capacity: The maximum workload that a system can handle without affecting performance.
  - Availability: The percentage of time a system is operational and available. This metric is often presented as a percentage, such as "99.9% uptime".

- Scalability metrics assess the system's ability to accommodate increased load.
- Reliability metrics measure how often a system experiences faults or failures. Mean Time Between Failures (MTBF) and Mean Time To Recovery (MTTR) are commonly used metrics.
- Cost metrics help businesses understand the economic implications of implementing and maintaining different architectures. This includes the cost of the resources used (like CPU, memory, storage, network bandwidth), the cost of licensing, maintenance costs, the cost of downtime, etc.

Each metric may have implications on the others. For instance, a system might achieve high throughput by using more resources, leading to higher costs. Similarly, striving for low latency might involve trade-offs in terms of cost and complexity. Therefore, these measurements should be seen as interconnected pieces of a larger puzzle that businesses must solve to find the optimal balance for their specific needs. Understanding and effectively utilizing these quantitative measurements is a key component of successful technology strategy in today's business landscape.

In order to evaluate our architectures, we use specific metrics, focusing on performance, that get recorded by Prometheus and get visualized in Grafana.

4.3 Message brokers

Transitioning our focus to the core components used in our architecture, we shall begin by exploring the message brokers.

A message broker is an application that acts as an intermediary or a middleware for communication between various applications. Message brokers offer two basic message distribution patterns or messaging styles [29]:

- Point-to-point messaging: This is the distribution pattern utilized in message queues with a one-to-one relationship between the message's sender and receiver. Each message in the queue is sent to only one recipient and is consumed only once.
- Publish/subscribe messaging: In this message distribution pattern, often referred to as "pub/sub," the producer of each message publishes it to a topic, and multiple

message consumers subscribe to topics from which they want to receive messages. All messages published to a topic are distributed to all the applications subscribed to it (broadcast-style method).

There are two approaches for message delivery. One is pull-based approach, where the subscriber queries the broker for new messages and the other is the push-based approach where the messages are automatically forwarded to the subscribers as soon as they are received. [30]

As we found out in the previous chapter, message brokers are used in asynchronous communication. An asynchronous communication system can be implemented in either one of two ways: one-to-one mode or one-to-many mode. In a one-to-one (queue) implementation there is a single producer and single receiver. But in one-to-many (topic) implementation there are multiple receivers and each request can be processed by zero to multiple receivers. [31]

4.3.1 RabbitMQ

Keywords: message broker, push-based, connection, channel, producer, consumer

RabbitMQ is an open-source message-broker software that implements the Advanced Message Queuing Protocol (AMQP) but also provides support for other protocols. The basic concepts of RMQ include the channel, the queue, the producer and the consumer. The implementation of a RabbitMQ application has the following steps: First, a connection is established to the RMQ Server, then a channel is created for the connection and finally, a queue is declared with a specific name. Once the application is up and running, the producer can send messages to the queue (publish function) and the consumer receives the messages from the specific queue by declaring the name of the queue (consume function).

The produced messages remain in the queue until they are handled by a consumer. By default, after a message is delivered, it will be deleted from the queue. It is also important to note that RabbitMQ is implemented in one-to-one mode, so each message is consumed by exactly one consumer. [32], [33]

RabbitMQ adopts a push policy for delivering messages to subscribers. This is not necessarily fast, especially when the message production rate is faster than the consumption rate, and harms performance when the queues are overloaded with messages. [30]

4.3.2 Redpanda

Keywords: event streaming platform, pub/sub, pull-based, producer, subscriber, topic

Redpanda is an event streaming platform: it provides the infrastructure for streaming real-time data. The basic concepts of Redpanda include producers, consumers and topics. Producers are client applications that send data to Redpanda in the form of events. Redpanda safely stores these events in sequence and organizes them into topics, which represent a replayable log of changes in the system. Consumers are client applications that subscribe to Redpanda topics to asynchronously read events, regardless of the events' publisher. Consumers can store, process, or react to the events. Producers and consumers interact with Redpanda using the Apache Kafka API. [34], [35]

Redpanda and Kafka share basic concepts. A server in a Kafka cluster (group of servers that operate together) is called a broker. Each topic is partitioned for scaling, parallelism and fault-tolerance. The typical consumer will process the next message in the list, although it can consume messages sequentially starting from any offset, as the Kafka cluster retains all published messages for a configurable period of time. Producers are able to choose which topic, and which partition within the topic, to publish the message to. Consumers assign themselves a consumer group name, and each message is delivered to one consumer within each subscribing consumer group. If all the consumers have different consumer groups, then messages are broadcasted to each consumer. Redpanda follows a one-to-many asynchronous communication implementation.

Still, Redpanda and Kafka have a number of differences. Compared to Kafka, Redpanda has significantly lower latency and higher performance, is much easier to install and tune and does not require using ZooKeeper (for management) or the JVM.

Redpanda employs a pull-based consumption model that allows an application to consume data at its own rate and rewind the consumption whenever needed. [36]

Redpanda comes with Redpanda Console, which is a developer-friendly web UI for managing and debugging your Kafka/Redpanda workloads. [37]

To get a better understanding of the key differences between RabbitMQ and Redpanda we present the following table. Keep in mind that these features may differ according to the examined scenario. [38], [39]

	RabbitMQ	Redpanda
Message delivery	push-based	pull-based
Messaging design	smart brokers / dumb clients	dumb brokers / smart clients
Implementation	one-to-one	one-to-many
Message Storage	queue	disk partition
Data persistence	when consumed, messages are deleted from queue	stores data for later use
Scalability	scales quite well	is highly scalable
Availability	increasing availability affects performance	high availability
Latency	lower end-to-end latency at lower throughputs	lower latency at higher throughputs

4.4 Other tools

Except for message brokers, various other tools were used in the development and testing of our application, all of which are presented below.

4.4.1 Docker

Keywords: OS-level virtualization, containers, environment consistency

Docker is a technology that allows you to build, run, test, and deploy distributed applications. It uses operating-system-level virtualization to deliver software in packages called containers. Each container is a standalone, executable package that includes everything needed to run a piece of software. This ensures that the software will always run the same, regardless of its environment thus reducing inconsistencies between development and production environments.

Docker is packaging an application and its dependencies in a virtual container that can run on any computer. In other words, a container encapsulates an application along with its runtime dependencies into a highly portable, standard image. This allows for applications to be decoupled from the operating system and underlying hardware, providing an environment where an application can be packaged once and run anywhere. More specifically, containers enable each workload to hold exclusive access to resources such as processor, memory, service account, and libraries, which is essential for the development process. They also run as groups of isolated processes within an operating system, allowing them to start fast and maintain. [15]

With regards to providing abstraction and isolation, containers closely resemble virtual machines (VMs); however unlike VMs, in a container the application is packaged with only the essential elements for it to run and not the entire guest OS. This allows containers to

have unique capabilities when compared with VMs like quick spin-up, minimum overhead, miniscule footprint, and high portability. [17]

Containers are a type of technology that has gained popularity due to their high performance, light weight, and enhanced scalability. They have proven particularly beneficial for microservices architecture, wherein applications consist of many independent services. [10] The lightweight characteristics of Docker containers can help creating and running more microservices, which furthermore contributes to a higher resource utilization. Also, containers can be easily moved to more performant machines. [40]

4.4.2 Prometheus

Keywords: monitoring system, time series database, real-time metrics

Prometheus is an open-source monitoring system with a dimensional data model, flexible query language, efficient time series database and modern alerting approach. It is primarily used for recording real-time metrics and it can generate insightful graphs, tables, and alerts. In other words, prometheus is able to collect all the metrics from a target system over HTTP and place it into a time-series database. It is a simple yet effective data model which when combined with a powerful query language, is able to inspect how the applications and underlying infrastructure is behaving. [17], [41]

One of the metric types offered by Prometheus is the histogram, which samples observations (usually things like request durations or response sizes) and counts them in configurable buckets. It also provides a sum of all observed values.

4.4.3 Grafana

Keywords: monitoring platform, queries, visualizations, alerts, graphs

Grafana is an open-source platform for monitoring and observability. It allows you to query, visualize, alert on, and understand your metrics. Grafana provides the users with tools to turn their time-series database data into beautiful graphs and visualizations. It has a data source model which is highly pluggable and supports multiple time-series–based data sources like Prometheus, InfluxDB, and OpenTSDB as well as SQL databases like MySQL and Postgres. Grafana is probably the only tool which supports combining data from many different sources into a single dashboard. [17]

4.4.4 PostgreSQL

Keywords: object-relational, DBMS, SQL

PostgreSQL is an advanced, enterprise class open source relational database that supports both SQL (relational) and JSON (non-relational) querying. It is a highly stable database management system, backed by more than 20 years of community development which has contributed to its high levels of resilience, integrity, and correctness. PostgreSQL is used as the primary data store or data warehouse for many web, mobile, geospatial, and analytics applications. [42]

4.4.5 Visual Paradigm

Keywords: UML, BPMN, modeling, agile development

Visual Paradigm is a software application designed for software development teams to model business information systems and manage development processes. In addition to modeling support, this technology provides report generation and code engineering capabilities including code generation. This technology can reverse engineer diagrams from code, and provide round-trip engineering for various programming languages. Visual Paradigm features Unified Modeling Language (UML) diagrams, Entity Relationship Diagram (ERD), and Object Relational Mapping Diagrams (ORMD) utilities essential in system and database design. [43]

4.4.6 Python

Keywords: programming language, high-level, scripting, agile development

Python is a high-level, interpreted programming language known for its clear syntax and readability. It supports both procedural and object-oriented programming paradigms and comes with a large standard library that includes areas like internet protocols, string

operations, web services tools, and operating system interfaces. Python's dynamic typing and built-in data structures make it an excellent language for scripting and rapid application development. Python's ecosystem is enriched by a thriving community which contributes to the development and maintenance of countless packages for diverse applications. The language's flexibility and versatility make it a powerful tool for any software development project.

Chapter 5

Case Study

To really test and appreciate the performance and the features of the different architectures, implementing them in a real application can be a good idea. This section presents the case study that our application relies on and proposes five different architectural approaches to be implemented and tested under its conditions.

Case Study: Alternative Microservice Implementations of a Data Warehouse Engine

- Overview: This case study explores the use of microservices architecture in the development of a data warehouse engine, focusing on parsing, analyzing and storing energy consumption data from the ENTsoe V3 API, for a number of countries, aiming at extracting data that are useful and easily accessible.
- Background: The European Union (EU) comprises multiple countries, each with its unique energy consumption patterns, which necessitates a sophisticated data analysis approach.
- API overview: The provided API, ENTsoe V3, presents a wealth of information regarding energy consumption for each country, timestamped every 15, 30 or 60 minutes, extending back to August 24, 2014.
- Goal: The task is to process this big data volume, extracting the daily, weekly, and monthly energy load for each EU member.

5.1 Functional requirements

The functional requirements of this project include:

1. Extraction of energy load data from the API.
2. Segregation of the data on a country-wise basis.
3. Aggregation of the energy load data on a daily, weekly, and monthly basis for each country.

It should be noted that this thesis does not go any deeper in the field of data analysis and thus we are not going to report the processed energy load for each country nor will we focus on the time it took for each worker to write the data to the database.

5.2 Non-functional requirements

The non-functional requirements include:

1. Performance: The system should perform data extraction, management, and reporting in an efficient and timely manner.
2. Scalability: The system should be scalable to accommodate future increases in data volume and frequency of API calls.
3. Reliability: The system should reliably access the API and handle any potential errors in the data or API downtimes.
4. Accuracy: The system should accurately calculate and report the aggregated energy consumption.
5. Maintainability: The architecture should be maintainable, with clear, modular code that allows for easy updates and debugging.

5.3 Workers

Within our software architectures, we deploy various types of specialized services, the workers, each performing specific background tasks. The workers can be classified into five different categories:

- Initiator worker: Upon the initialization of our application, the initiator worker is responsible for the creation of database tables and possibly the establishment of the connection with our message brokers, RabbitMQ or Redpanda. This ensures that the system is properly set up before the main processing begins.
- Data worker: The function of a data worker is to fetch data from a specified API via a GET request, parse the retrieved data, and relay it to the remaining workers. Given the considerable volume of data processed, we have designed multiple data workers, each assigned to a specific country (e.g., data-worker-germany, data-worker-spain). In total, our system hosts 27 dedicated data workers, effectively segregating data handling per country.
- Daily worker: This worker calculates the daily load and stores the result to the database.
- Weekly worker: This worker calculates and stores the weekly load.
- Monthly worker: This worker calculates and records the monthly load.

In our architectures, the number of daily, weekly and monthly workers can be specified by the user of the application. All daily, weekly and monthly workers have the same number of instances (considered as N in the UML diagrams). For example, if the user

specifies the number to 3, there will be 3 daily workers, 3 weekly workers and 3 monthly workers, working in parallel.

The communication mechanism between the workers differs according to the architecture in use, it could involve RMQ queues, a Redpanda topic or websockets. The workers also interact with other components (such as the database) in the system to accomplish their tasks.

Load balancing

One thing that needs to be pointed out before describing our architectures, is the way that messages are distributed when multiple daily, weekly or monthly workers run in the system. In the case of architectures that use message brokers, it is quite straightforward as the brokers handle the distribution of the messages. In order to achieve this, RabbitMQ uses a round-robin algorithm and Redpanda uses consumer groups.

However, in other architectures, the way that we distribute the messages is that we assign each (of the 27) data worker to a specific daily/weekly/monthly worker based on the modulo of the data worker's index (0 to 26) to the number of parallel workers (1 or 3). What this effectively does is evenly distribute the workers across different hosts in a round-robin fashion. For example, in case of 3 parallel workers, the first data worker connects to daily-worker-0, the second to daily-worker-1, the third to daily-worker-2 and then the fourth data worker goes back to daily-worker-0, and so on. This helps to balance the load across all available data workers.

Chapter 6

Architectures

As mentioned previously, our project includes the development of five different architectures. In this chapter we are going to introduce each one and explain how the various components work together to achieve their goals. Since all of the tools used have already been presented in the previous chapter, we are now focusing on their collaboration with each other and the final setup. It is vital to note that the architectural solutions we present are built using Python as our programming language, since a different stack could produce a different system behavior.

6.1 Architecture descriptions

All architectures have a few things in common, so we will begin by looking at these. Common features:

- All of our workers (Initiator, Data, Daily, Weekly and Monthly) run in a Docker environment in separate containers. The initiator worker is responsible for creating the database tables for the total, daily, weekly and monthly load to be stored.
- Our PostgreSQL database is connected to Docker via a TCP connection and is running in port 5432.
- Data workers interact with the provided API through an HTTP GET request method. This interaction is facilitated via the endpoint 'api/v3/data'. Each worker gets only the data for the specific country it is responsible for. This way, the data workers now have the total load data specified for the country of interest.
- Data workers parse the data in batches of one month, so each request includes one month's data.
- Prometheus follows a pull-based model where it scrapes metrics from the workers over HTTP. It is integrated with all the workers via the HTTP endpoint 'http/metrics'. Prometheus runs in port 9090.
- Grafana is connected to Prometheus via an HTTP connection and Prometheus acts as a data source providing Grafana with the metrics. Grafana runs in port 3000.

- The number of our Data workers is the same and is equal to 27, as many as the EU countries given in the data.

All of the above mentioned information, as well as some more details, can be clearly understood by looking at the UML component diagrams of each architecture. These include all of the components while also stating the connection or communication between them. Moreover, another view of each architecture is described through the UML sequence diagrams that show how the components (lifelines) interact with each other based on the time sequence.

Now, we are ready to look further into each one of the architectures.

6.1.1 Orchestrator

This architecture utilizes websockets for the communication between the workers. The data workers act as the central control unit that orchestrates the application processes. In further detail, each data worker adheres to the following pattern:

- The data worker sends the total load to the daily worker that processes the information and returns the daily load, again through a websocket connection. The daily worker also saves the daily load to the db.
- Then, the data worker sends the daily load to the weekly worker that processes the data and returns the weekly load. The weekly worker also saves the weekly load to the db.
- Finally, the data worker sends the weekly load to the monthly worker that processes the information and saves the monthly load to the db.

The UML component diagram can be found below.

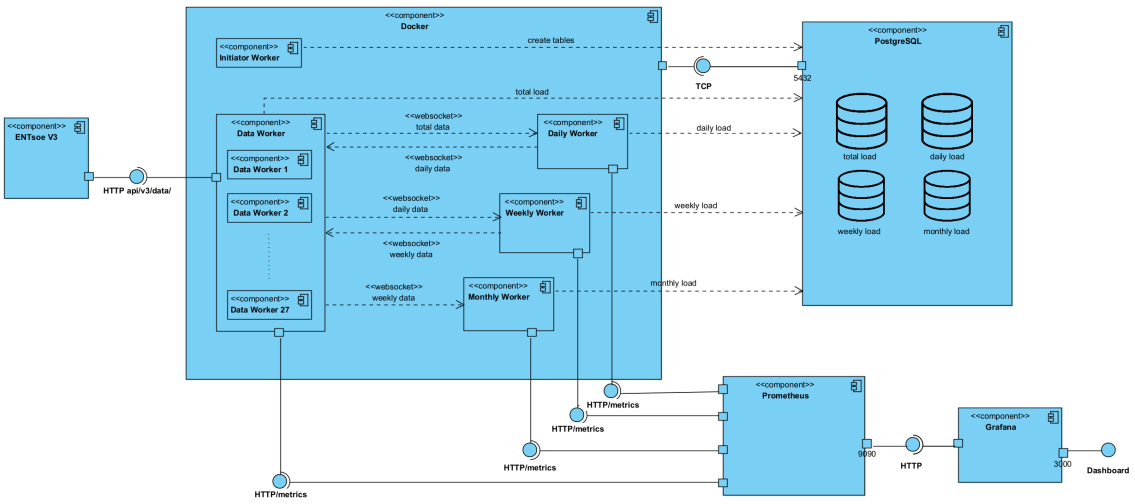


Figure 6.1: *Component diagram of the Orchestrator architecture*

The UML sequence diagram is the following.

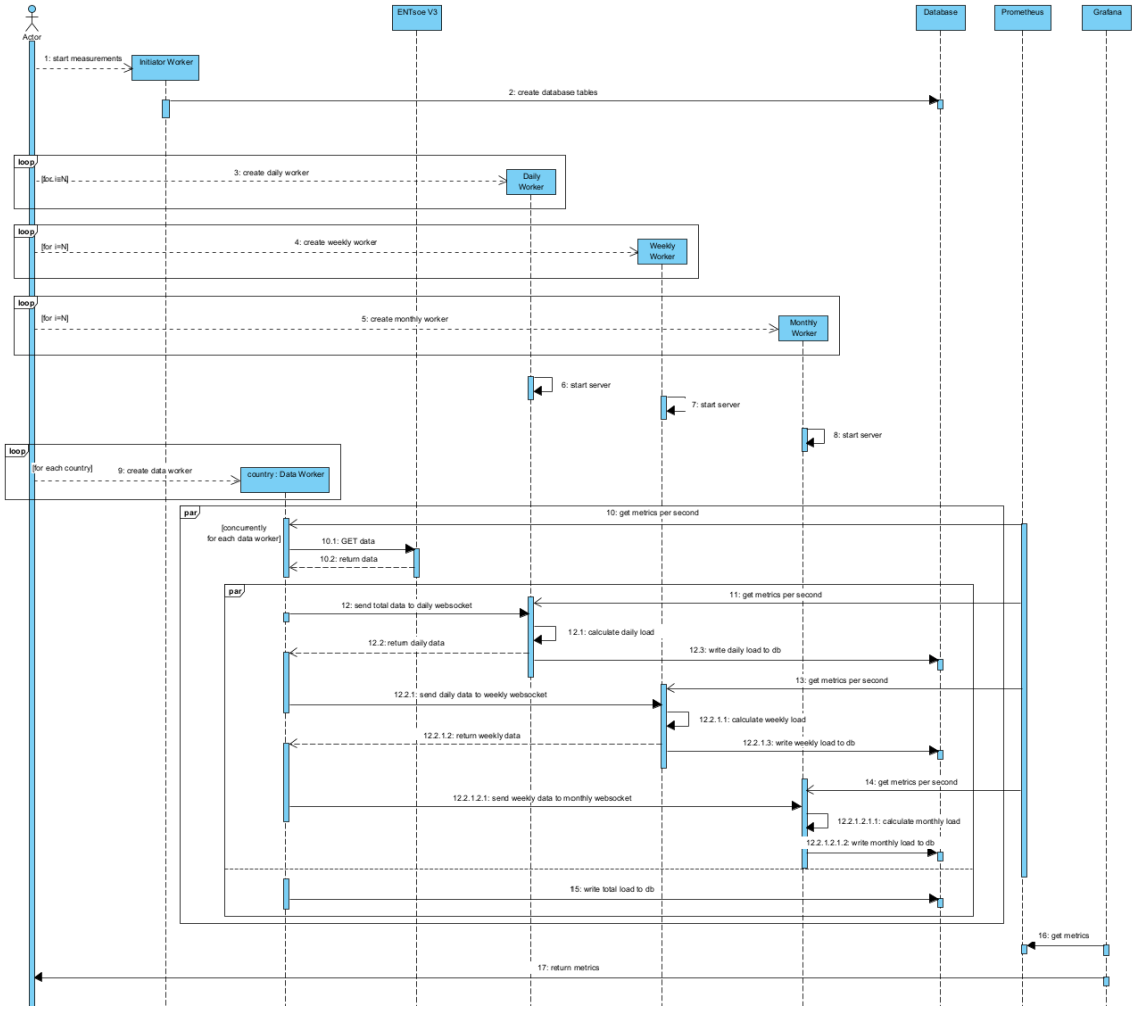


Figure 6.2: *Sequence diagram of the Orchestrator architecture*

6.1.2 Serialised Orchestrator

This architecture also utilizes websockets for the communication between the workers. However, the sequence of events is more straight-forward. The data worker sends the total load to the daily worker which in turn sends the daily load to the weekly worker which sends the weekly load to the monthly worker. Each of the workers writes their processed data to the database.

The UML component diagram is the following.

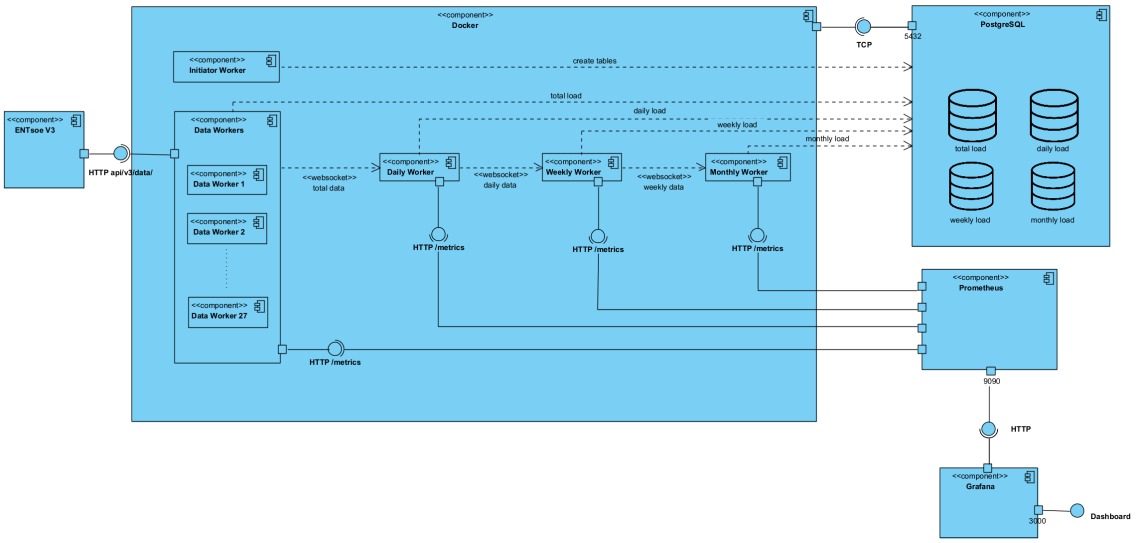


Figure 6.3: *Component diagram of the Serialised Orchestrator architecture*

And the UML sequence diagram is below.

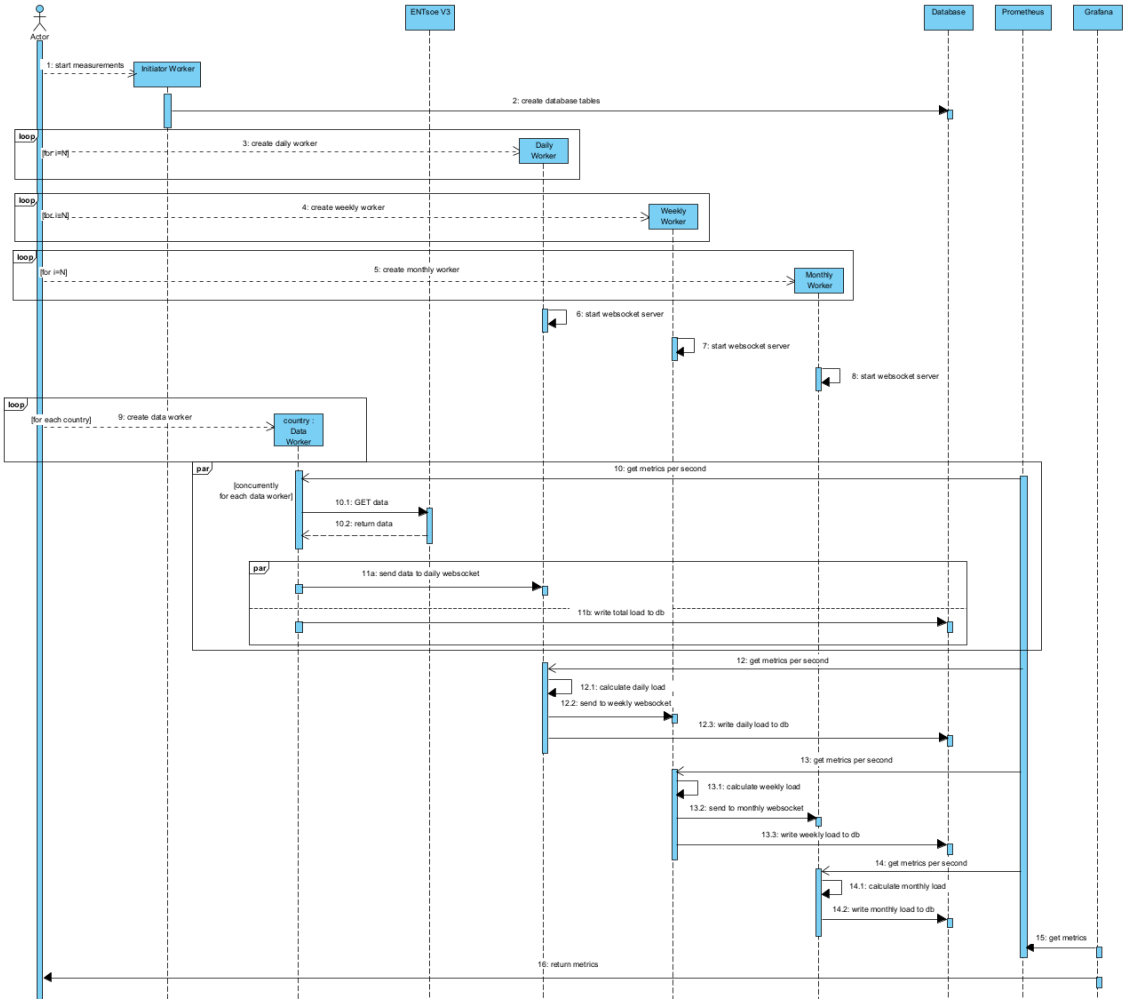


Figure 6.4: *Sequence diagram of the Serialised Orchestrator architecture*

6.1.3 Async Orchestrator

This architecture also utilizes websockets for the communication between the workers. This time, the data worker sends the total load to all daily, weekly and monthly workers at the same time (asynchronously). This means that weekly and monthly workers have to process the data in a different way than before since they are getting the total load (and not the daily or weekly load respectively). Each of the workers writes their processed data to the database.

The UML component diagram is:

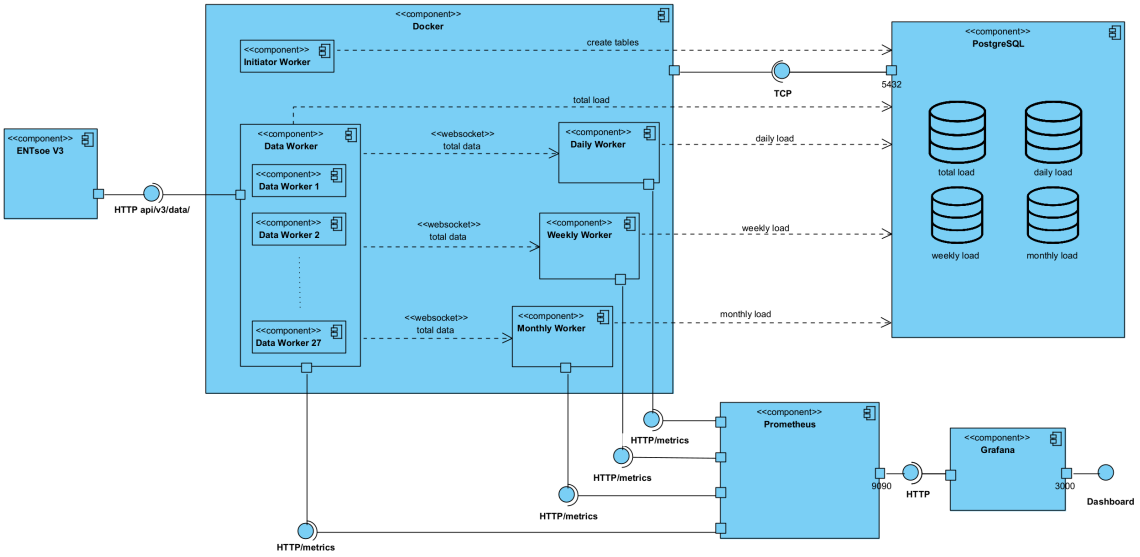


Figure 6.5: *Component diagram of the Async Orchestrator architecture*

And the sequence diagram is the following:

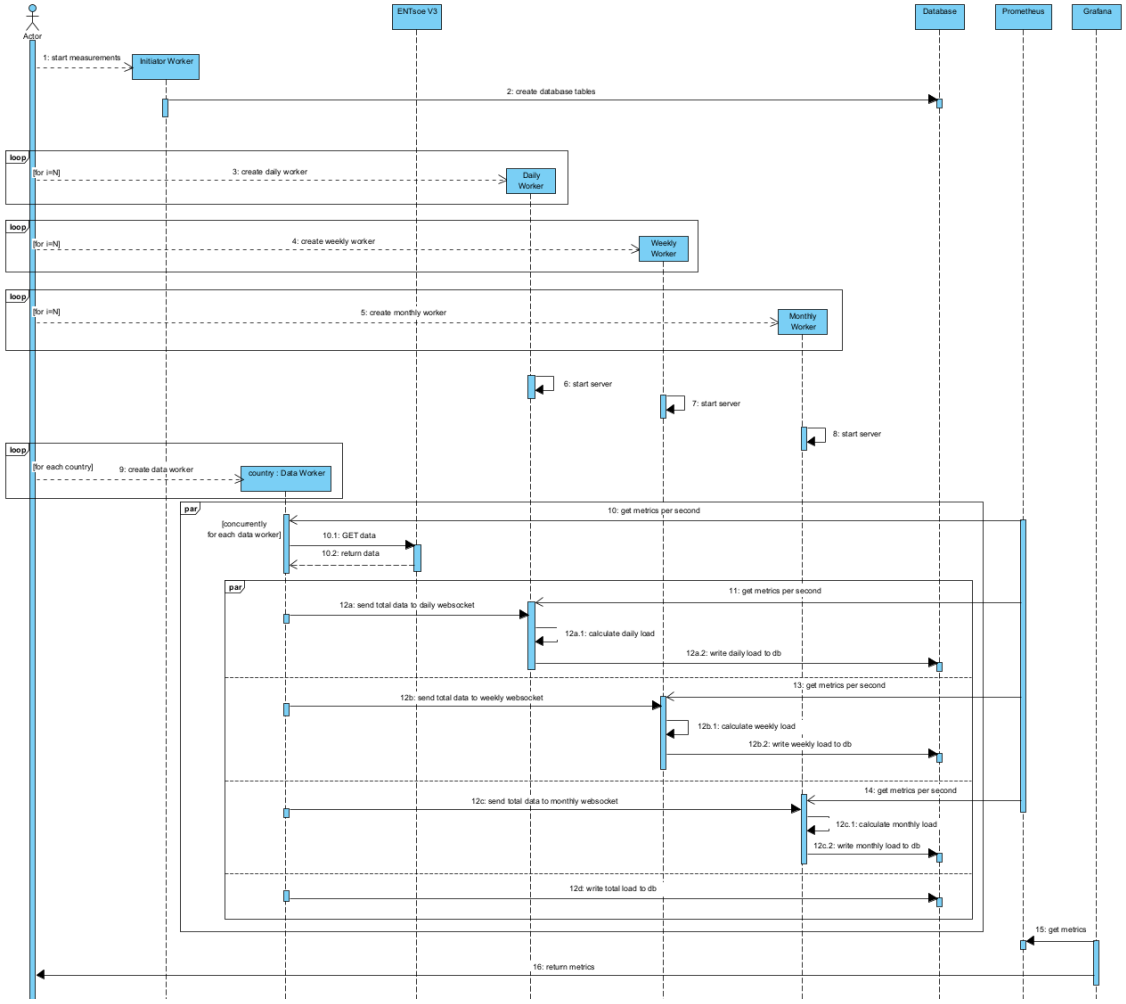


Figure 6.6: *Sequence diagram of the Async Orchestrator architecture*

6.1.4 Serialised RabbitMQ

In this architecture the communication is established with the use of the RabbitMQ message broker. RabbitMQ is connected to Docker via a TCP connection and is running in port 15672. The Initiator worker has the extra role of initializing the RMQ daily, weekly and monthly queues. Each queue is named after the receiver of the messages. In further detail, the process goes as follows:

- The data worker adds the total load to the daily queue.
- The daily worker receives the total load (from the daily queue), calculates the daily load and adds it to the weekly queue.
- The weekly worker receives the daily load (from the weekly queue), calculates the weekly load and adds it to the monthly queue.
- Finally, the monthly worker receives the weekly load (from the monthly queue) and calculates the monthly load.

All workers also save their processed data to the database, in the corresponding tables. The UML component diagram gives a detailed idea of the component-level structure.

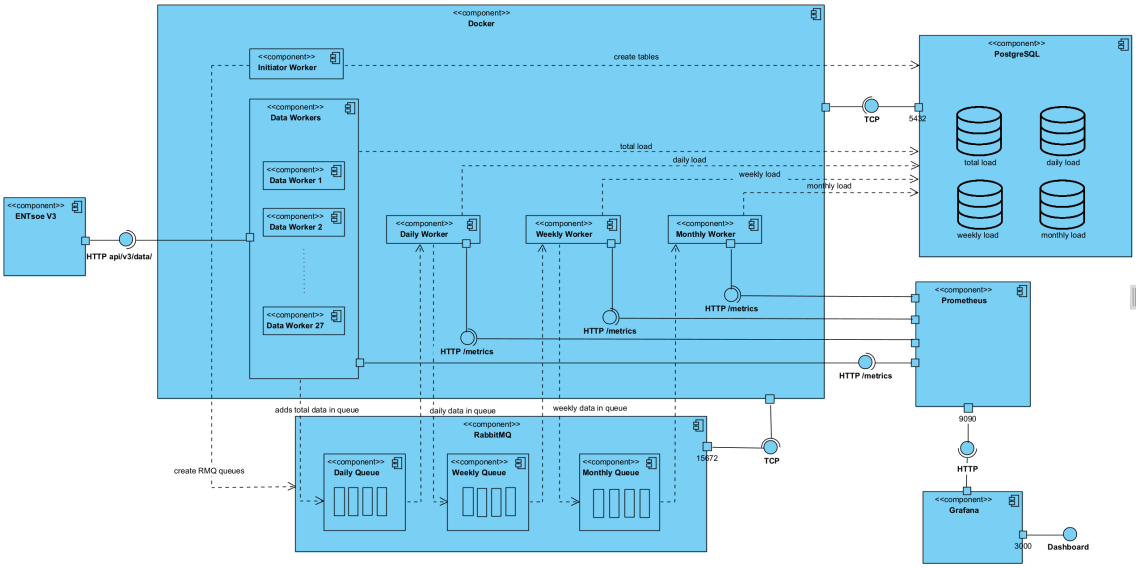


Figure 6.7: *Component diagram of the Serialised RabbitMQ architecture*

And the UML sequence diagram is the following.

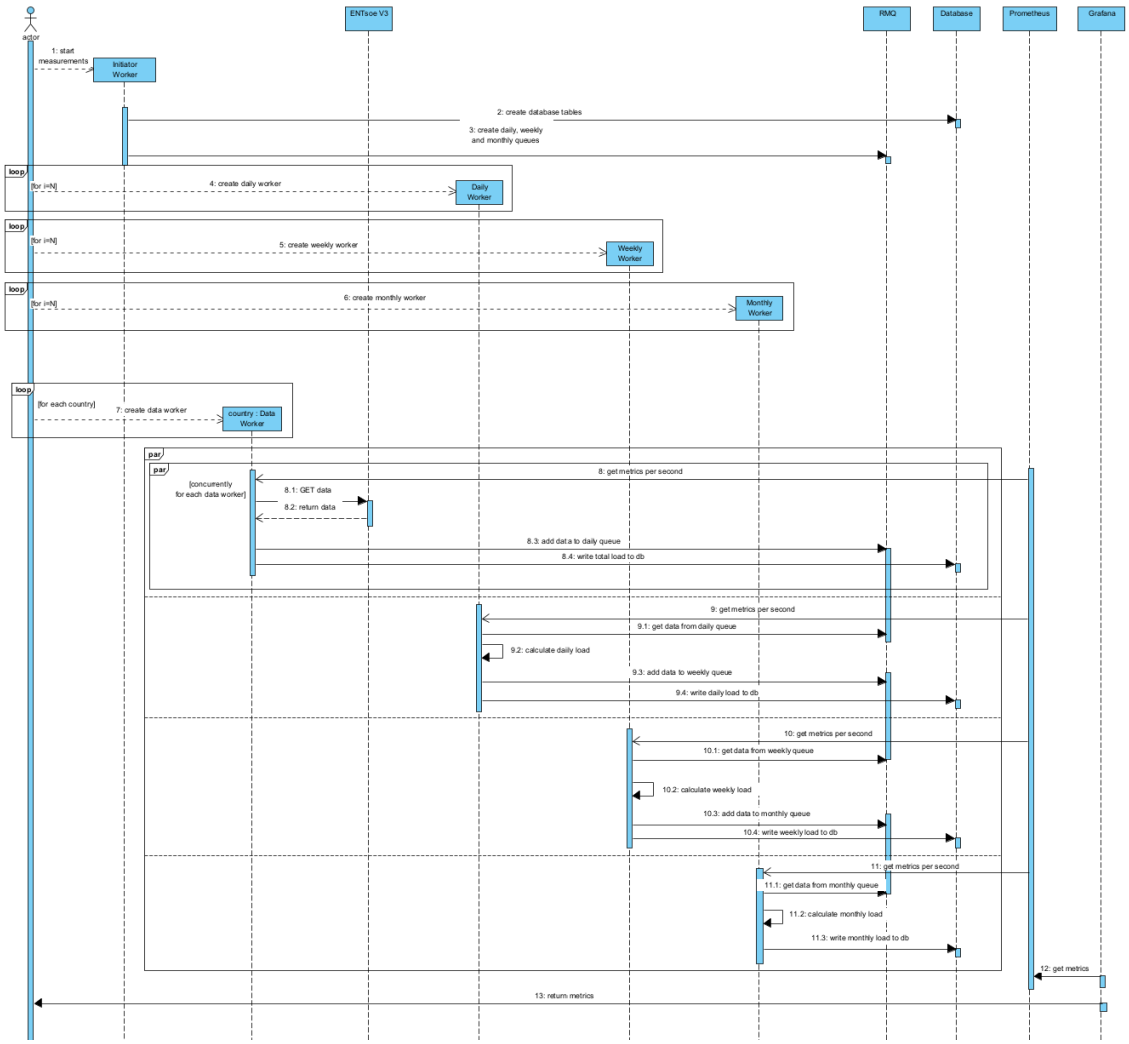


Figure 6.8: *Sequence diagram of the Serialised RabbitMQ architecture*

6.1.5 SOA Redpanda

In this architecture the communication is established with the use of the Redpanda broker. Redpanda is connected to Docker via a TCP connection and is running in port 18081. The Initiator worker, here, has the extra role of initializing the Redpanda topic. The data worker produces data to the 'General topic' within Redpanda and all daily, weekly and monthly workers subscribe to that topic in order to receive the messages. The total data is consumed simultaneously by daily, weekly and monthly workers.

All workers after processing the data also save them to the database, in the corresponding tables.

The UML component diagram is the one below.

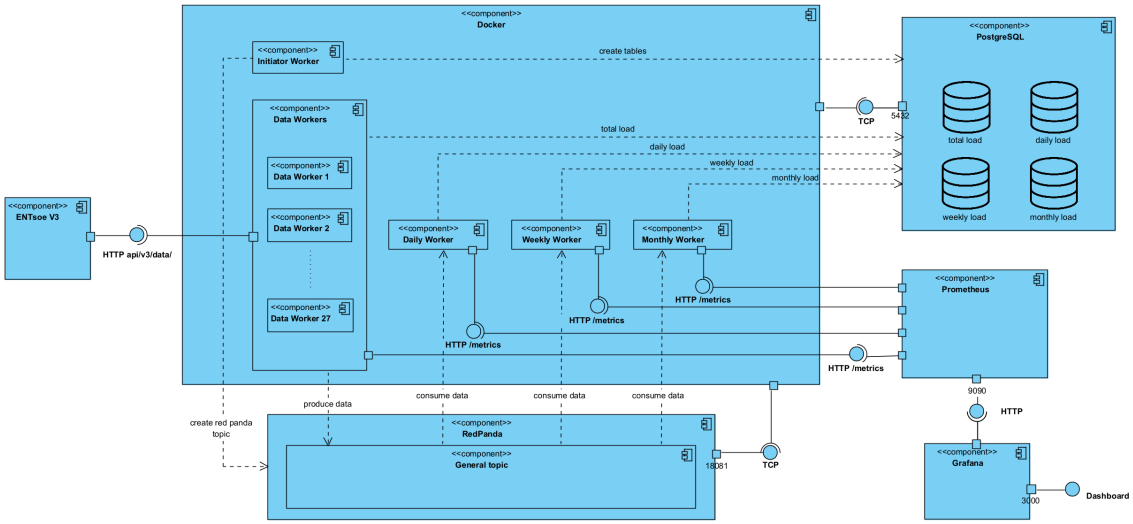


Figure 6.9: *Component diagram of the SOA Redpanda architecture*

The UML sequence diagram is the following.

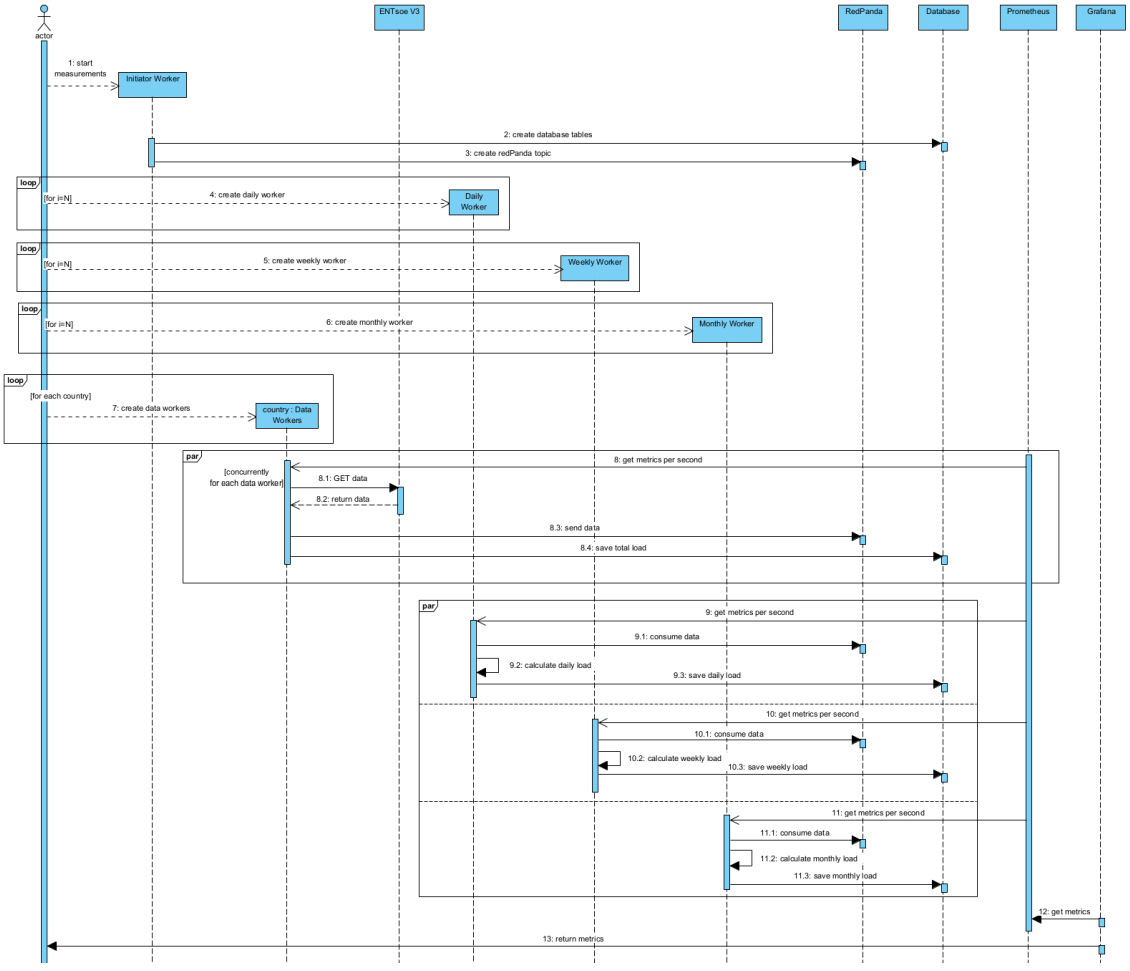


Figure 6.10: *Sequence diagram of the SOA Redpanda architecture*

6.2 System deployment

The deployment of the system architecture is a crucial stage of the project. It dictates how various components interact with each other, determines system performance, and significantly influences the system's resilience and scalability. The deployed architecture for this project incorporates 4 different servers, each with distinct roles and specifications, thereby allowing for an efficient allocation of tasks and resources.

- Front server: 2 CPU cores, 4GB RAM, 40GB of disk space

  The Front server houses Metabase and Grafana - two critical tools used in our data visualization pipeline.

- Broker Server: 4 CPU cores, 16GB RAM, 40GB of disk space

  In case the architecture includes a message broker (either RMQ or Redpanda), then this is located in the Broker server. The substantial computational resources on this server ensure smooth message brokerage and effective communication between different system components.

- DB Server: 2 CPU cores, 8GB RAM, 40GB of disk space

  The DB server hosts the PostgreSQL database. The hardware resources for this server are dedicated to ensuring swift database operations and data integrity.

- Workers Server: 4 CPU cores, 16GB RAM, 40GB of disk space

  The Workers server hosts all the workers and Prometheus. The hardware resources of this server are tuned to ensure efficient data processing and system monitoring.

The exchange of data between our servers and the data workers, as well as the overall system communication, is fortified through the use of a Virtual Private Network (VPN).

In addition to the above descriptions, we have created several deployment diagrams to provide a visual representation of the architectures. These diagrams further elucidate the interaction between different components and the overall structure of the system.

UML Deployment diagrams

For all Orchestrator architectures built on Websockets, there is no Broker Server and the following deployment diagram applies:

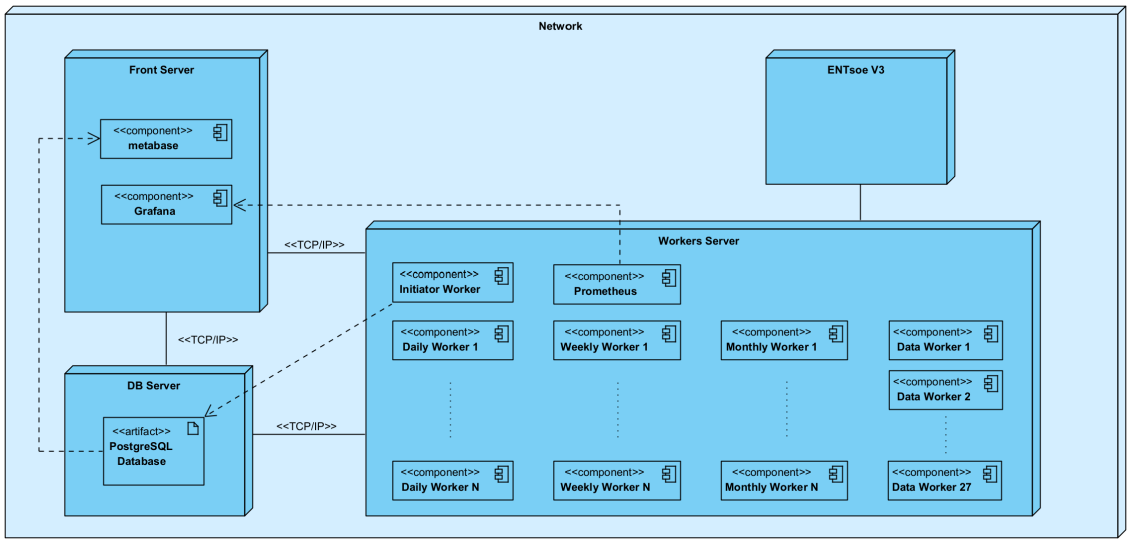


Figure 6.11: *Deployment diagram of the Orchestrator, Async Orchestrator and Serialised Orchestrator architectures*

For Serialised RMQ the deployment diagram, including the RabbitMQ in the Broker Server, is below.

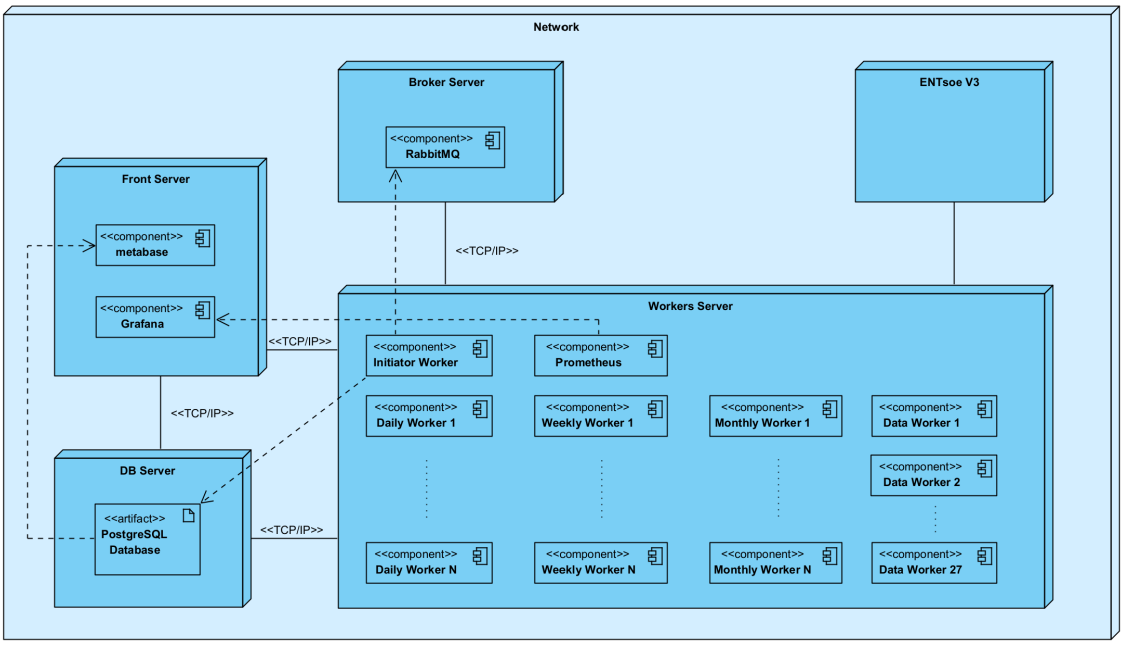


Figure 6.12: *Deployment diagram of the Serialised RabbitMQ architecture*

For SOA Redpanda the deployment diagram, including the Redpanda in the Broker Server is the following. In this architecture, Front Server also includes the Redpanda console.

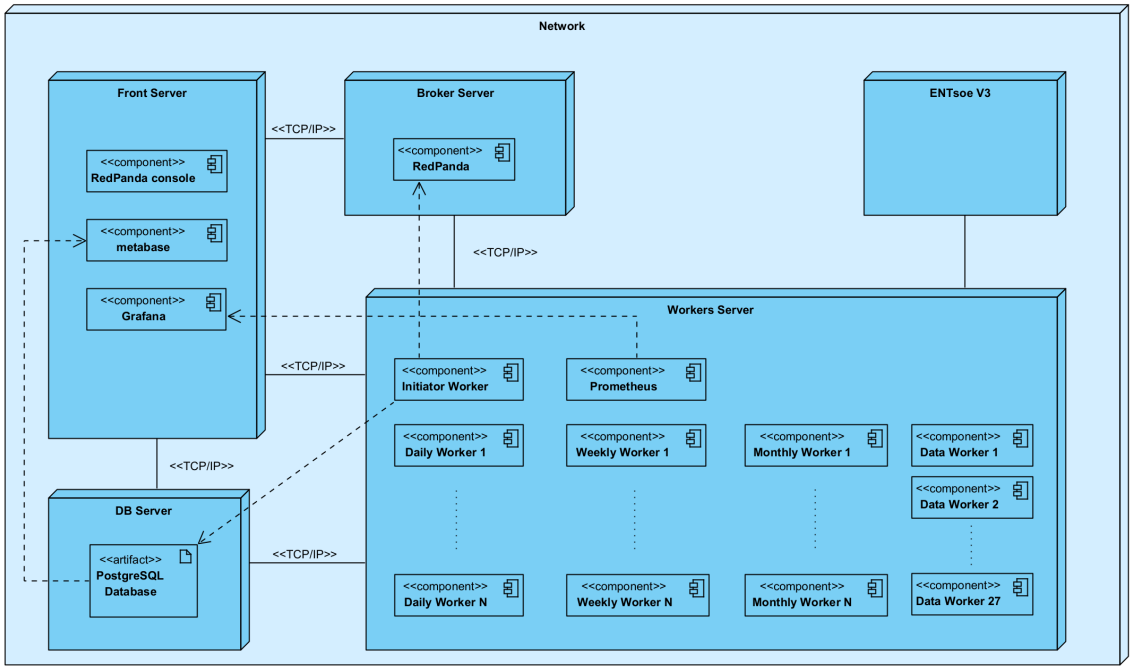


Figure 6.13: *Deployment diagram of the SOA Redpanda architecture*

Chapter 7

Benchmarking and Results

In this section, we are going to present our metrics, the tests that were conducted and the results as well as some of the observations that we made based on them.

7.1 Metrics

In the development stage of the application we decided to record a number of different metrics in order to later be able to analyze the behavior of our system and each architecture. The following is a list of these metrics:

Cross Worker Metrics

- Time spent from Request to Monthly data write: measures the total time elapsed from receiving a data request to writing the monthly data to the database.
- Average time spent from Request to each Worker: measures the time from when a data request is received to when it is processed by each worker.
- Min and Max time spent from Request to each Worker

Metrics for Data Worker

- Time for a 30-day request: measures the time taken to process a data request (each request spans 30 days).
- Time to save in DB: measures the time taken by a worker to save processed data into the database.
- Total Requests Processed: tracks the total number of data requests processed by a worker.
- Total Bytes Received: keeps track of the total volume of data handled by a worker.
- Average Bytes Per Request: provides an average size of the data requests that a worker processes (total bytes received / total requests processed).

Metrics for Daily, Weekly and Monthly workers

- Total Requests Processed

- Total Bytes Received
- Average Bytes Per Request
- Time to parse data
- Time to write data to DB
- Time to consume and forward data: measures the time taken to consume data from the broker and forward it for further processing or storage.

Of course, not all metrics in this list produced interesting results in the testing phase, so we will subsequently focus on the metrics that give us more information on the comparison of our five architectures.

In order for the reader to better understand the environment we worked in and the visualization of the metrics, we provide a screenshot of the Grafana dashboard taken at the time of the testing phase.

Figure 7.1: *Grafana dashboard at the time of testing*

7.2 Execution and difficulties

7.2.1 Tests

System testing forms the backbone of any reliable software deployment strategy, offering essential insights into the performance, scalability, and stability of the implemented solution. In this project, a series of comprehensive tests were conducted using a variety of configurations and data volumes. This allowed us to evaluate the performance and scalability of our microservice-based data warehouse architecture under different load conditions.

To keep in mind, we need to mention again that our system works with 27 data workers, one for each country and that their number remained unchanged throughout the testing phase.

For each architecture we performed the following tests:

- Daily, weekly and monthly workers: 1 of each and 3 of each

  We started with testing the performance of the system with a single worker of each kind (daily, weekly, monthly). We then simulated the processing of data for periods of:

  - 1 year of data (since 1st May 2022)
  - 3 years of data (since 1st May 2020)
  - 6 years of data (since 1st May 2017)

  We continued with increasing the number of workers to three of each (daily, weekly, monthly) and performed the same tests for 1 year, 3 years and 6 years.

In order to better explain the testing phase we present the below diagram.

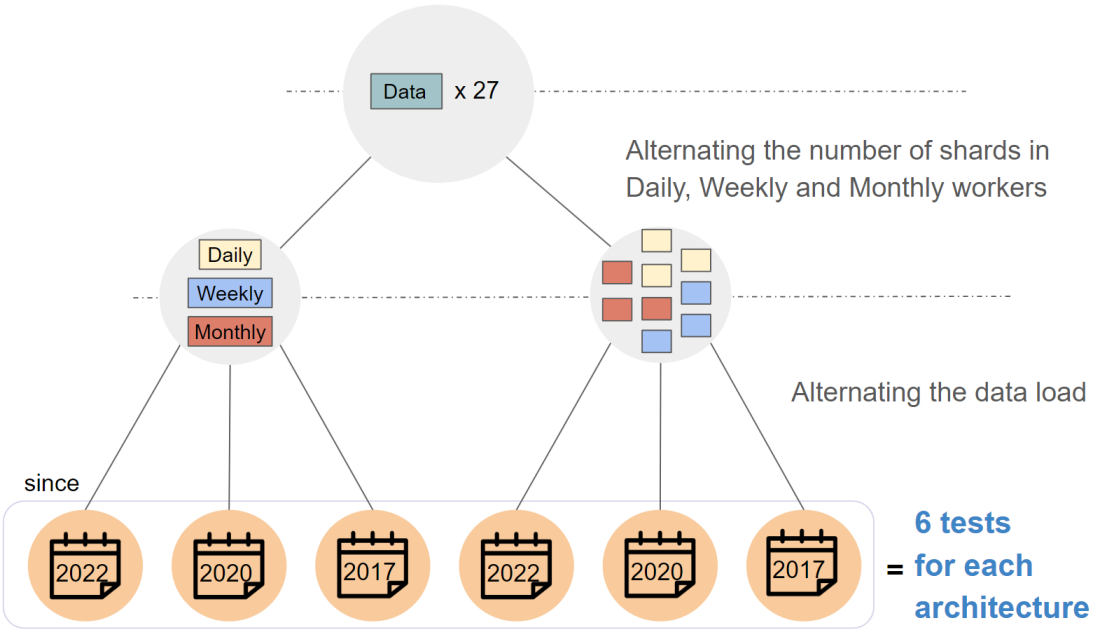


Figure 7.2: *Visual representation of the testing*

7.2.2 Running the tests

We continue by talking about the execution of the tests. The same tests (as mentioned in the previous paragraph) were performed for each architecture under the same conditions.

A few things about the testing need to be noted before we move on to their execution. First, we should mention that all data workers were coordinated to start at the same time. So, although the data workers were created sequentially, for each country, they were programmed to wait for a specific interval so that it is ensured they will all start simultaneously. Another important element is that at the beginning of each test the database was empty. All tables were deleted after each test and then recreated in the next one. Finally, the workers' VM was cleared up after each testing, which means that all workers were shut

down and rerun in the next test. All of the above actions were needed to ensure that no conditions differed between the tests.

In order to perform the tests, the setup we built was this:

- We connected to a VPN
- We connected to each of our 4 VMs, hosting each one of our servers (front, broker, workers and database)
- We got all of the external services up and running - Grafana, Prometheus, Docker, Postgres.
- We got our workers running

Afterwards, we started testing our application and all of our architectures one by one. Initially, we had only 1 of each of the daily, weekly and monthly workers and tested for 1 year, 3 years and 6 years of data. Then, we increased the number of workers to 3 of each and ran the app with the same configurations.

7.2.3 Problems and observations

One problem faced during the testing of our application is that Docker gradually exhausted disk space due to some leakage. To solve this, we had to regularly use the 'docker prune' command to delete all images and volumes of containers. By running these commands after the testing of each architecture and regularly checking our disk space, we were able to control the docker leakage.

Another important problem that hindered the testing phase was that Prometheus consumed great network bandwidth. This was because Prometheus was scraping the workers very frequently (every 1 sec) and so we had to increase the scrap_interval (to 5 sec) mentioned in the prometheus.yml file.

One thing we observed from looking into the metrics of our system was how the weekly worker was slower than the others. We suspect this is because of the function that calculates the week of the data based on the data, but further investigation into this should be carried out.

Once the testing phase was done and judging by the metrics we got and the diagrams that were made, we realized that our implementation of the orchestrator architectures was wrong so we looked into it. The problem was lying on the way the websockets connection was implemented so we fixed that and rerun all of the tests.

The final results are listed in the next section.

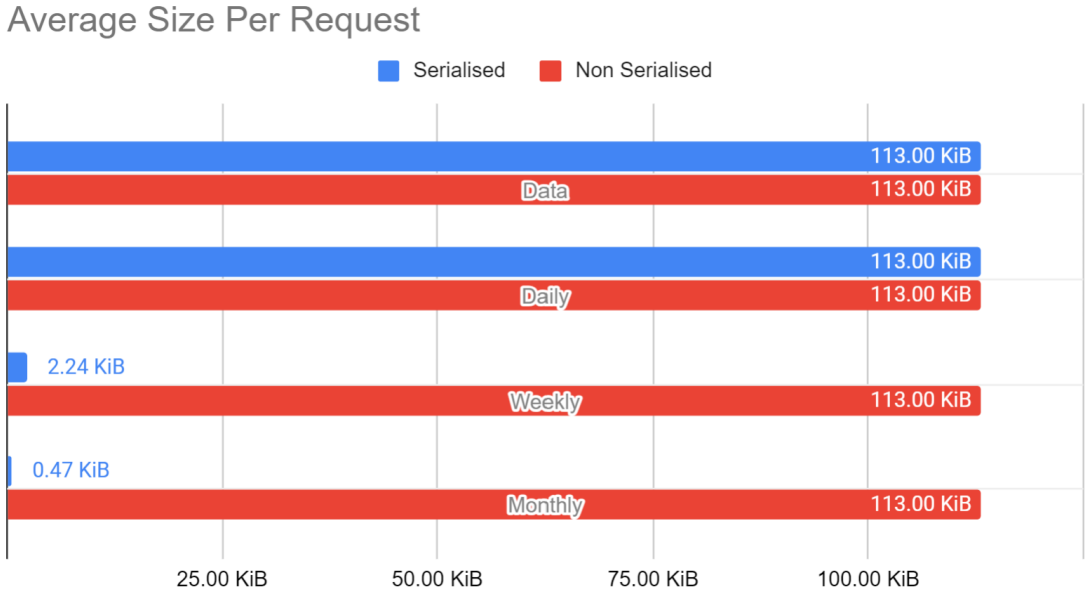
7.3 Results

Each of the tests revealed a variety of metrics for our system. In order to understand these metrics and explain the results in a simple way, we continued by making various diagrams that better compare the behavior of our architectures.

Before presenting the results we need to mention a few potentially useful information for the overall evaluation of the system. These have to do with the time spent for each request in the API and also the amount of data that we get from it.

- Average Time Per Request in ENTsoe V3: 1s
- Total Requests at 1 year: 405
- Total Requests at 3 years: 1053
- Total Requests at 6 years: 2025
- Total Bytes at 1 year: 44MB
- Total Bytes at 3 years: 118MB
- Total Bytes at 6 years: 224MB

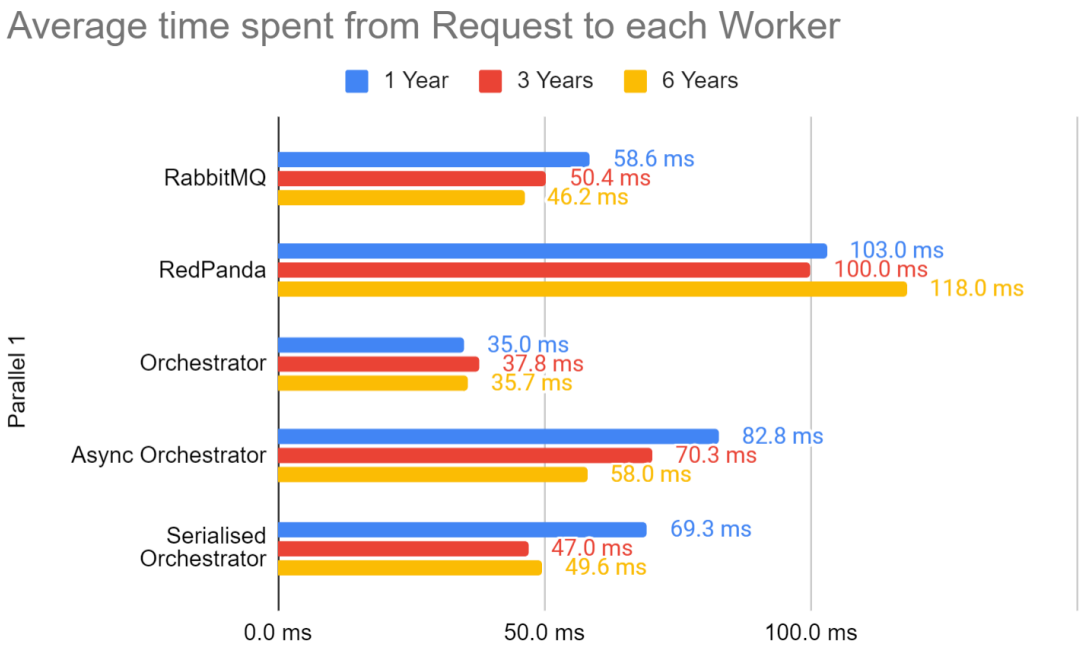
Another thing we would like to point out before looking at the time it took for each architecture to run is the difference in the size of the request that each worker receives. On that note, we decided to compare serialized with non serialized architectures in account of the size of the request they receive. Of course, since in non serialized architectures all workers get the total load, they all get the same amount of data. However, in serialized architectures, the average size of the request that the weekly and monthly workers receive had to be smaller. Indeed, by looking at the diagram, we can see that the size of the request in serialized architectures has dropped dramatically for the last two workers, reaching 470 Bytes for the monthly worker to process.



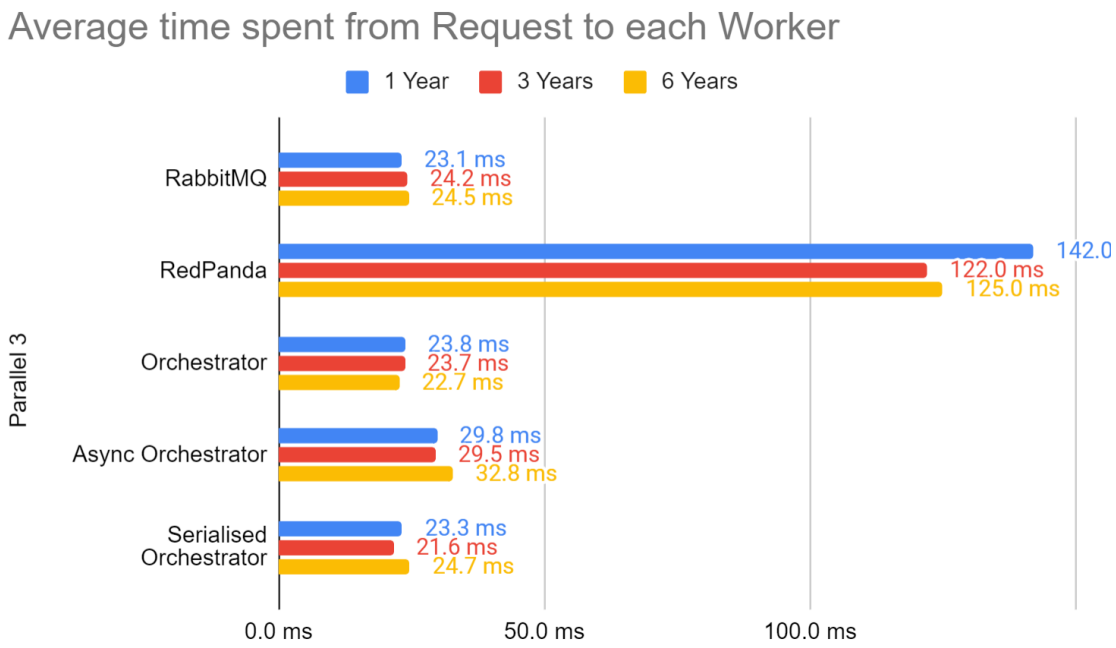
Now, we can go ahead on monitoring the average time it takes from the request to each worker. This means the metric that starts at the time when the data worker has received the response of a request to the API and stops right after another worker (daily, weekly or monthly) saves their processed data in DB.

From this metric we decided to present the time of the slowest worker in two diagrams, as depicted below. The first diagram shows the metric for 1 worker, of each (daily, weekly, monthly). In this diagram, we made the following observations.

- In some architectures, the time drops while the load increases (for one to three years). This is probably because the older the data the sparser. Another reason would be that all workers are coordinated to start at the same time, so there is great load at the point the system starts while later on the average value of the message delay lowers.
- Redpanda is the slowest architecture. The reason behind this lies in the fact that Redpanda implements a REST API connection, so consumers perform GET requests to get the data. The exchange of messages to open and close the connection is costly. Another cause of why Redpanda is slow is that it performs multiple jobs such as keeping the data for later consumption and being responsible for the resiliency of data.
- Async Orchestrator is slower than the other Orchestrator architectures. One cause of this is that each message that is received for the weekly and daily workers is large since all workers get the total load.
- Orchestrator architecture is the fastest. This is because orchestrator implements a single connection between the workers while in a serialised orchestrator two connections are established each time (one connection to the previous worker and one to the next)

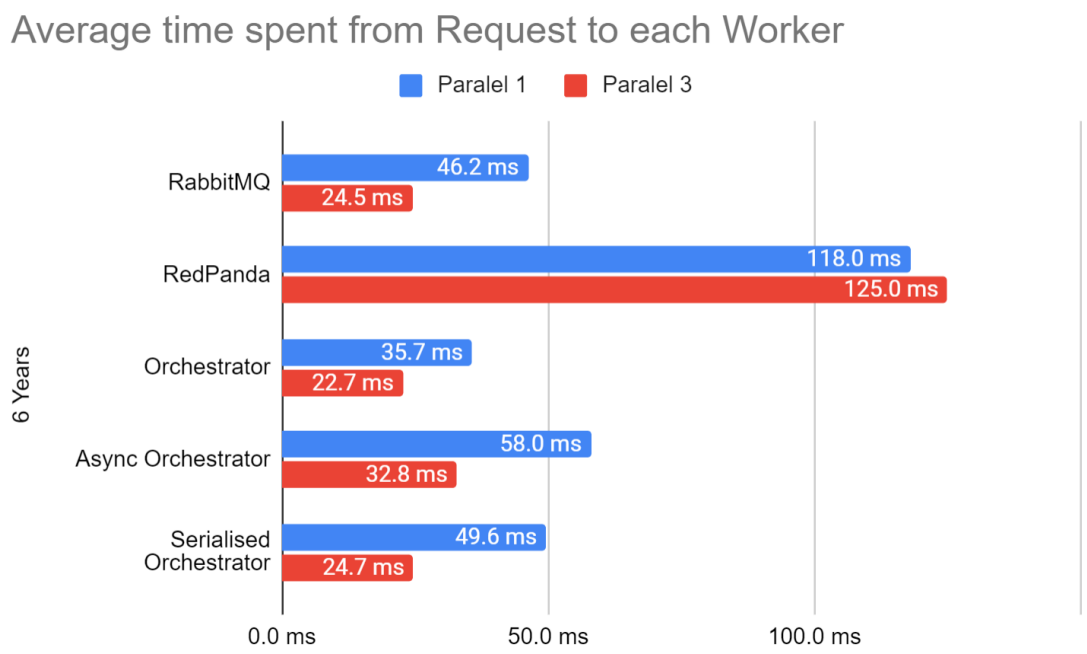


The second diagram shows the metric for 3 workers, of each (daily, weekly, monthly) working in parallel.



The first thing we can clearly examine from the above graph is that parallelism dramatically enhanced the average time from request to each worker. Another thing we can observe is that most architectures have similar metrics. This is due to the fact that all connection are TCP which means that they are keepalive (allows connection to remain active even when no data is exchanged for a long period). The architectures get bottlenecked by the nature of the connection, thus they all metrics are similar, and not by the load of the data. Finally, we can see that RabbitMQ has similar performance to the Orchestator architectures and this is because they are all push-based, so they have the same communication system. Finally Redpanda is still the slowest architecture due to its resiliency, it consumes more system resources thus reducing the performance.

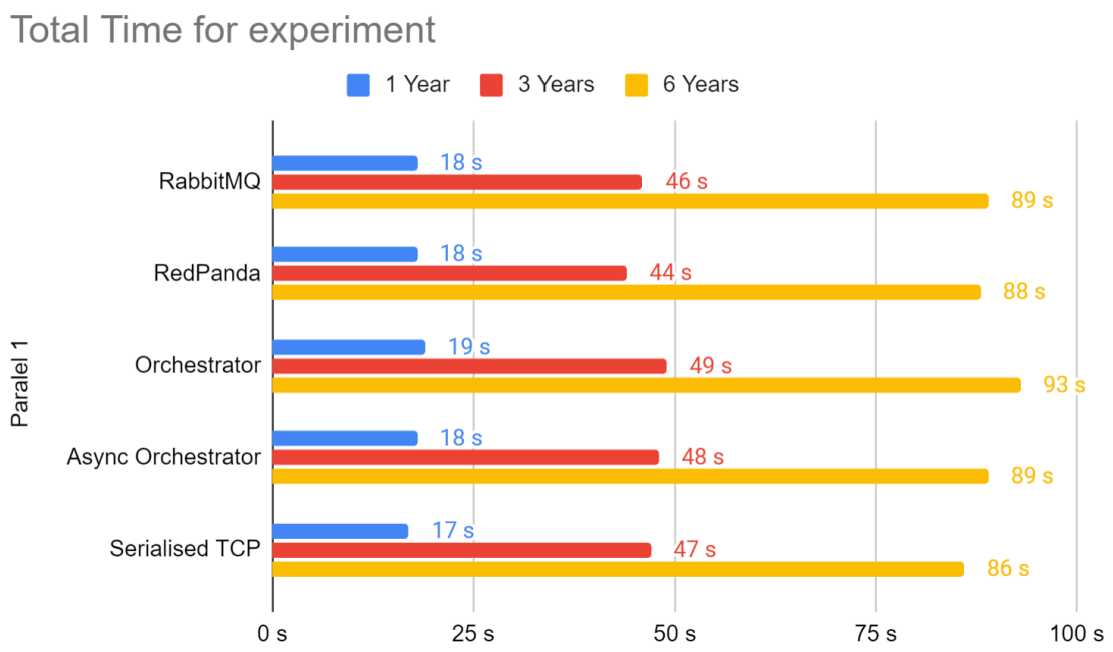
In the next diagram we decided to keep the data we got from the previous metric and compare the 1 worker and 3 workers for each architecture. The reason behind this graph is to show how parallelism affects the system more clearly.



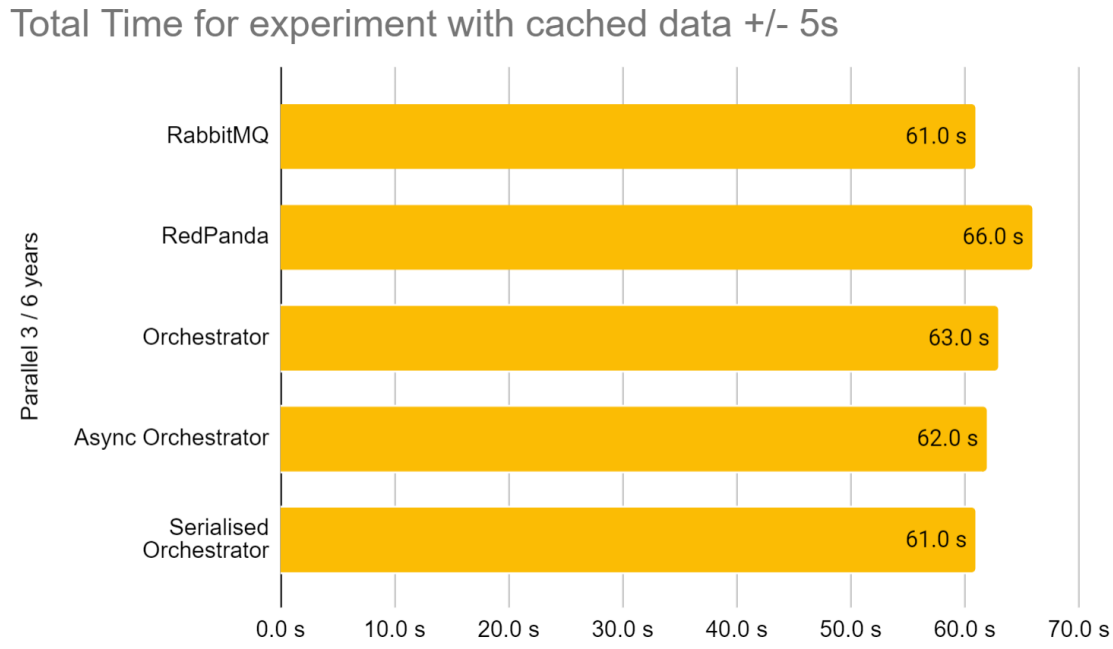
It seems that because of peer-to-peer connections, parallelism of workers significantly improved the time.

Proceeding to the next graph, we compare the total execution time of each architecture. In this graph, it is obvious that no distinctive differences appear, so all architectures complete the requirement in the same amount of time. This is because this time our API (ENTsoe) acts as a bottleneck, since each request takes 1 sec. The question that comes to mind is 'Why are our architectures important then?'. The answer to this question is that the goal of our system is to make the data available to the consumers and to get them fast.

The fact that the total execution time of all architectures is the same makes this metric valid. This is because if each architecture's time was different then each one would also have a different CPU time (so more available resources) and we would not be able to compare the architectures.



Finally, we decided it would be interesting to test the system under abnormal conditions and observe its behavior. In this regard, we moved all of our data (from ENTsoe V3) locally and then ran the system so that we make all requests simultaneously. The below graph shows the total execution time of the system under heavy load.



From the above diagram we observe two things. First, we have a new bottleneck that is the general capacity of our system. At the time of getting the metrics we checked the CPUs of the system and they were all full. Second, we have to mention that since the system's capacity cannot cope with the time it takes for the data to get processed, then

the architecture of the system is irrelevant.

The source code for the application discussed in this thesis is provided in Appendix B for further review and analysis. The results of the tests are also available in the form of snapshots from Grafana, in Appendix A.

Chapter 8

Conclusions

In this concluding chapter, we aim to give a comprehensive assessment of various architectural performances based on the measured metrics and interpreted graphs provided in Section 7.3. It is crucial to underscore that the objective of this study is to explore and understand the distinct performances of our architectures rather than making definitive conclusions about them.

8.1 Comparison of architectures

Redpanda, a powerful open-source alternative to Apache Kafka, is ideal for handling big data systems with a high volume of producers and consumers that demand statelessness. This system's strength lies in its flexibility, specifically the ability to set any consumer offset. This unique feature allows for the possibility to replay data in various scenarios, enhancing the system's resilience and ensuring system continuity during unexpected events. Further, Red Panda's instant scalability becomes essential when it comes to managing workloads effectively and efficiently, as it allows for quick and seamless addition of workers as and when required. But one must consider that Redpanda's strengths come with the trade-off of increased infrastructure demands. Therefore, for latency-sensitive applications where minimizing per-request delays is vital, Redpanda might not be the optimal choice.

RabbitMQ is a robust open-source message broker that excels in situations where data consumption prioritization is crucial. Its built-in First-In-First-Out (FIFO) queue ensures prioritized and orderly data processing, a feature especially beneficial in systems where message sequence is important. Beyond prioritization, RabbitMQ's reputation as a high-performance message broker stems from its stability, flexibility, and compatibility with multiple messaging protocols. Also, its capability for instant scalability, made possible by its support for clustering and high-availability configurations, is a valuable feature for dynamic and growing systems. Despite these advantages, it's important to recognize that RabbitMQ may not be the best choice when message consumption prioritization is not a top priority, or when dealing with very high throughput rates as it might impact performance.

An Orchestrator shines when you require a centralized control plane to oversee multiple independent services, particularly in a microservices architecture. Its capacity to streamline processes, ensure system-wide consistency and harmonize communication between services can significantly improve system performance. Additionally, the Orchestrator offers a level

of reliability, making sure that if a single service fails, it does not affect the overall system. However, while an Orchestrator provides many benefits, it also maintains a level of state awareness. This can introduce complexity when handling failures and limit its applicability in systems that need to operate in a stateless manner.

The Async Orchestrator leverages asynchronous communication, a method of communication where the sender and receiver do not need to interact with the message at the same time. This model allows tasks to be handled independently and concurrently, effectively utilizing available CPU resources, and greatly reducing communication costs. It's particularly useful in situations where a system has surplus CPU resources and needs to handle multiple communications efficiently without blocking or waiting. However, this asynchronous model does have its limitations. For example, if a system is CPU-bound and doesn't have extra resources, the asynchronous operations could lead to performance issues due to the overhead of context switching and potential difficulties in debugging and handling errors.

Finally, a Serialized Orchestrator is designed to process tasks in a serial or sequential manner. This can be particularly useful in systems where metadata processing can significantly enhance performance, or where the sequence of task execution is critical. The serialized processing model ensures a defined order of operations, maintaining the integrity and accuracy of processed data. This can be especially beneficial in complex systems where tasks have dependencies or order of execution matters. However, it's essential to consider that a Serialized Orchestrator may not be the best fit for systems where tasks are independent and can be processed concurrently, as it could potentially underutilize available resources and lead to decreased overall system performance.

8.2 Further improvements

Firstly, the exploration of alternative software stacks such as Golang or Rust can potentially lead to different outcomes, thus enriching our perspective on the varying impacts of distinct programming environments. This variation might pave the way for optimized choices based on specific requirements or constraints, ultimately enhancing the performance and efficacy of our technological solutions.

Secondly, further scrutinizing our initial conclusions through rigorous validation processes presents an excellent opportunity for contributing to the expanding research around microservices. It is through this in-depth investigation that we can either confirm or debunk our findings, thereby refining our understanding and ensuring that our conclusions are grounded in robust evidence.

Lastly, employing a combination of various communication methodologies can significantly enrich our investigation. This comprehensive approach enables us to gain a nuanced perspective on how different interaction mechanisms affect system performance and reliability. Such an understanding is invaluable in facilitating more efficient and reliable system designs, further pushing the boundaries of what can be achieved within the realm of microservices.

Appendix A

Grafana Snapshots

All of the metrics recorded by Prometheus and Grafana can be found in a visual representation within the Grafana dashboard. In the following links you can take a look at the results.

For 1 of each workers (daily, weekly, monthly):

1 worker	**1 Year**	**3 Years**	**6 Years**
RabbitMQ	RMQ1-one	RMQ1-three	RMQ1-six
Redpanda	Redpanda1-one	Redpanda1-three	Redpanda1-six
Orchestrator	Orch1-one	Orch1-three	Orch1-six
Async Orchestrator	AsyncOrch1-one	AsyncOrch1-three	AsyncOrch1-six
Serialised Orchestrator	SerialOrch1-one	SerialOrch1-three	SerialOrch1-six

For 3 of each workers in parallel:

3 workers	**1 Year**	**3 Years**	**6 Years**
RabbitMQ	RMQ3-one	RMQ3-three	RMQ3-six
Redpanda	Redpanda3-one	Redpanda3-three	Redpanda3-six
Orchestrator	Orch3-one	Orch3-three	Orch3-six
Async Orchestrator	AsyncOrch3-one	AsyncOrch3-three	AsyncOrch3-six
Serialised Orchestrator	SerialOrch3-one	SerialOrch3-three	SerialOrch3-six

After experimenting with cached data in order to increase the system's load:

with cached data	**6 Years**
RabbitMQ	RMQ-cached
Redpanda	Redpanda-cached
Orchestrator	Orch-cached
Async Orchestrator	AsyncOrch-cached
Serialised Orchestrator	SerialOrch-cached

Appendix B

Source Code

For those wishing to examine our project in more detail, we provide you with the link to the GitHub repository where the source code of the application for this thesis is available.

Link to the repository: https://github.com/ntua/apistresstesting

Keep in mind that the link will be accessible by all viewers once the repository gets published.

Bibliography

[1] Davide Falessi, Giovanni Cantone, Rick Kazman και Philippe Kruchten. *Decision-making techniques for software architecture design: A comparative survey*. *ACM Computing Surveys*, 43(4):1–28, 2011.

[2] Len Bass, Paul Clements και Rick Kazman. *Software architecture in practice*. Addison-Wesley, Upper Saddle River, NJ, third editionη έκδοση, 2013. OCLC: 825819423.

[3] David Garlan. *Software Architecture*. `https://www.nasa.gov/pdf/637608main_day_2-david_garlan.pdf`, 2012.

[4] Felix Bachmann, Len Bass, Jeromy Carriere, Paul Clements, David Garlan, James Ivers, Robert Nord και Reed Little. *Software Architecture Documentation in Practice: Documenting Architectural Layers:*. Τεχνική Αναφορά με αριθμό, Defense Technical Information Center, Fort Belvoir, VA, 2000.

[5] David Garlan. *Software architecture: a roadmap*. *Proceedings of the Conference on The Future of Software Engineering*, pages 91–101, Limerick Ireland, 2000. ACM.

[6] P. Clements, D. Garlan, R. Little, R. Nord και J. Stafford. *Documenting software architectures: views and beyond*. *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 740–741, Portland, OR, USA, 2003. IEEE.

[7] Paul C Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord και Judith Stafford. *A Practical Method for Documenting Software Architectures*.

[8] Visual Paradigm. *What is Unified Modeling Language (UML)?* `https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/`.

[9] Mohamad Kassab, Manuel Mazzara, JooYoung Lee και Giancarlo Succi. *Software architectural patterns in practice: an empirical study*. *Innovations in Systems and Software Engineering*, 14(4):263–271, 2018.

[10] Guozhi Liu, Bi Huang, Zhihong Liang, Minmin Qin, Hua Zhou και Zhang Li. *Microservices: architecture, container, and challenges*. *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 629–635, Macau, China, 2020. IEEE.

[11] Antonio Bucchiarone, Nicola Dragoni, Schahram Dustdar, Stephan T. Larsen και Manuel Mazzara. *From Monolithic to Microservices: An Experience Report from the Banking Domain*. *IEEE Software*, 35(3):50–55, 2018.

[12] Francisco Ponce, Gaston Marquez και Hernan Astudillo. *Migrating from monolithic architecture to microservices: A Rapid Review*. *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–7, Concepcion, Chile, 2019. IEEE.

[13] ParTech Media. *A to Z of Microservices*. `https://www.partech.nl/nl/publicaties/2020/12/a-to-z-of-microservices`, 2020.

[14] Mark Richards. *Microservices vs. Service- Oriented Architecture*. O'Reilly Media, Inc, 2016.

[15] Freddy Tapia, Miguel Ángel Mora, Walter Fuertes, Hernán Aules, Edwin Flores και Theofilos Toulkeridis. *From Monolithic Systems to Microservices: A Comparative Study of Performance*. *Applied Sciences*, 10(17):5797, 2020.

[16] Ervin Djogic, Samir Ribic και Dzenana Donko. *Monolithic to microservices redesign of event driven integration platform*. *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1411–1414, Opatija, 2018. IEEE.

[17] Mainak Chakraborty και Ajit Pratap Kundan. *Monitoring Cloud-Native Applications: Lead Agile Operations Confidently Using Open Source Software*. Apress, Berkeley, CA, 2021.

[18] *Microservice Architecture - Blueprint*. `https://www.tutorialspoint.com/microservice_architecture/microservice_architecture_blueprint.htm`.

[19] Sam Newman. *Building microservices: designing fine-grained systems*. O'Reilly Media, Beijing Sebastopol, CA, first editionη έκδοσηη, 2015. OCLC: ocn881657228.

[20] *What is Microservices Architecture? | Microservices Definition*. `https://katalon.com/resources-center/blog/microservices-introduction`.

[21] Chaitanya K. Rudrabhatla. *Impacts of Decomposition Techniques on Performance and Latency of Microservices*. *International Journal of Advanced Computer Science and Applications*, 11(8), 2020.

[22] Wilhelm Hasselbring και Guido Steinacker. *Microservice Architectures for Scalability, Agility and Reliability in E-Commerce*. *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 243–246, Gothenburg, Sweden, 2017. IEEE.

[23] Morgan Bruce και Paulo A. Pereira. *Microservices in Action*. 2019.

[24] Javad Ghofrani και Daniel Lübke. *Challenges of Microservices Architecture: A Survey on the State of the Practice*. *ZEUS 2018*, Dresden, Germany, 2018.

[25] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin και Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*. *Present and Ulterior Software Engineering*Manuel Mazzara και Bertrand Meyer, επιμελητές, pages 195–216. Springer International Publishing, Cham, 2017.

[26] Filip Bahnan. *A Comparison Between the Quality Characteristics of Two Microservice Applications*. 2021.

[27] Alan Megargel, Christopher M. Poskitt και Venky Shankararaman. *Microservices Orchestration vs. Choreography: A Decision Framework*. *2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC)*, pages 134–141, Gold Coast, Australia, 2021. IEEE.

[28] *Service-level Requirements | What Is System Architecture? | InformIT*. `https://www.informit.com/articles/article.aspx?p=29030&seqNum=5`.

[29] *What are Message Brokers? | IBM*. `https://www.ibm.com/topics/message-brokers`.

[30] Apostolos Lazidis, Euripides G. M. Petrakis, Spyridon Chouliaras και Stelios Sotiriadis. *Open-Source Publish-Subscribe Systems: A Comparative Study*. *Advanced Information Networking and Applications*Leonard Barolli, Farookh Hussain και Tomoya Enokido, επιμελητές, τόμος 449, pages 105–115. Springer International Publishing, Cham, 2022. Series Title: Lecture Notes in Networks and Systems.

[31] Mehmet Ozkaya. *Microservices Communications*. `https://medium.com/design-microservices-architecture-with-patterns/microservices-communications-f319f8d76b71`, 2023.

[32] Aditya Jadon. *Understanding RabbitMQ Queue & Messaging Simplified 101 - Learn | Hevo*. `https://hevodata.com/learn/rabbitmq-queue/`, 2022. Section: message broker.

[33] Valeriu Manuel Ionescu. *The analysis of the performance of RabbitMQ and ActiveMQ*.

[34] *Introduction to Redpanda | Redpanda Docs*. `https://docs.redpanda.com/docs/get-started/intro-to-events/`.

[35] Ying Liu και Beth Plale. *Survey of Publish Subscribe Event Systems*.

[36] Khin Me Me Thein. *Apache Kafka: Next Generation Distributed Messaging System*. 2014.

[37] *Manage Redpanda Console | Redpanda Docs*. `https://docs.redpanda.com/docs/manage/console/`.

[38] Redpanda. *Redpanda as a RabbitMQ alternative*. `https://redpanda.com/blog/rabbitmq-alternative-redpanda-vs-rabbitmq`.

[39] *Benchmarking Kafka vs. Pulsar vs. RabbitMQ: Which is Fastest?* https://www.confluent.io/blog/kafka-fastest-messaging-system/.

[40] Thomas Schirgi. *Architectural Quality Attributes for the Microservices of CaRE*. 2021.

[41] Prometheus. *Overview | Prometheus*. https://prometheus.io/docs/introduction/overview/.

[42] *What is PostgreSQL? – Amazon Web Services*. https://aws.amazon.com/rds/postgresql/what-is-postgresql/.

[43] *What is Visual Paradigm*. https://www.oit.va.gov/Services/TRM/ToolPage.aspx?tid=10208.